Universidad Carlos III de Madrid
Escuela Politécnica Superior

Ingeniería en Informática

PROYECTO FIN DE CARRERA

Two Steps Reinforcement Learning en Robocup-Soccer Keepaway

AUTOR: Iván López-Bueno Hernández
DIRECTOR: Fernando Fernández Rebollo

LEGANÉS
Junio de 2009

Resumen

Hay una gran variedad de problemas en los que es necesario o muy deseable que un sistema aprenda de forma automática cómo debe interactuar con el entorno que le rodea para así, ser eficiente, eficaz y autónomo con respecto a la tarea que debe desempeñar. Por ejemplo, un problema de estas características puede ser un robot explorador que parta de una ubicación base y tenga que recolectar todas las muestras de mineral posibles en un terreno desconocido, teniendo que volver al punto base por el camino más corto antes de que sus baterías se agoten.

Las técnicas de aprendizaje por refuerzo permiten a un agente o sistema aprender cual es la mejor acción a ejecutar para llevar a cabo un objetivo cuando se encuentra en una situación determinada. Estas técnicas utilizan un enfoque en el que el agente es informado por el entorno acerca de si la última acción realizada fue buena o mala para conseguir el objetivo deseado. Esta información es proporcionada por el entorno en forma de señal, denominada señal de refuerzo y es empleada por el agente para aprender de forma eficiente un comportamiento que le permita llevar a cabo su objetivo de la mejor forma posible. Sin embargo, las técnicas de aprendizaje por refuerzo no son aplicables cuando el número de situaciones posibles o el número de acciones a tomar es excesivamente grande. En esos casos es necesario buscar alternativas de representación del conjunto de situaciones y acciones que simplifiquen los conjuntos y generalicen sus propiedades. En el presente documento se aborda el problema de la búsqueda de discretizaciones del espacio de estados que permitan una posterior aplicación de técnicas de aprendizaje por refuerzo sobre dichas discretizaciones. Se analizan y comparan distintas técnicas y se realiza una experimentación centrada en la técnica *Two Steps Reinforcement Learning*. El dominio utilizado como banco de pruebas de los experimentos realizados es el *Robocup-soccer keepaway*.

INDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

1. Motivación y objetivos

Un agente autónomo se define como aquél que interactúa con el entorno circundante por sí mismo. Debe poseer los sensores adecuados que le permitan obtener información importante proveniente del entorno (capacidad de percibir), debe poder convertir esa información adquirida en conocimiento que después pueda utilizar para lograr sus objetivos (capacidad de razonar) y además debe estar capacitado de actuar sobre el entorno en el que se encuentra (capacidad de actuación) [Moriello, 2005].

Si a este tipo de agentes se les da la capacidad determinar y recordar si una acción hecha ante una situación dada le fue favorable o no, estamos ante un agente inteligente autónomo [Fritz et al, 1990].

Para conseguir dotar de inteligencia a un agente artificial podemos programar de antemano el comportamiento del agente previendo cuales serán las situaciones en las que se encontrará y las acciones que ha de realizar en cada caso. En un segundo enfoque, podemos aplicar sobre el agente técnicas de aprendizaje que le permitan aprender el comportamiento que le hará llegar a cumplir su objetivo a través del la experiencia de la interacción del propio agente con el entorno. Una de estas técnicas que permiten dotar de capacidad de aprendizaje a un agente es el aprendizaje por refuerzo, basado en *Procesos de decisión de Markov* (MDP'S) [Sutton y Barto, 1998].

1.1 Motivación

Las técnicas de aprendizaje por refuerzo parten de que el problema a tratar está compuesto por un conjunto de estados y acciones finitos de tamaño manejable computacionalmente. Sin embargo, muchos de los problemas reales están compuestos por conjuntos de estados y acciones muy grandes o infinitos. *A priori*, en este tipo de problemas complejos, la aplicación de aprendizaje por refuerzo es inviable. Sin embargo, esta dificultad puede ser solventada buscando representaciones alternativas de los espacios de estados y acciones que contengan la información esencial que daban los originales. La búsqueda de representaciones alternativas que permitan la aplicación de técnicas de aprendizaje por refuerzo en dominios con espacios de estados y/o acciones grandes o infinitos está encuadrada dentro del marco de la denominada generalización en aprendizaje por refuerzo. Una de las técnicas que trata de solucionar el problema de generalización es la denominada 2SRL (*Two Steps Reinforcement Learning*) [Fernandez y Borrajo, 2008].

El algoritmo 2SRL combina la aproximación de funciones con técnicas clásicas de aprendizaje por refuerzo para conseguir obtener la política óptima en el dominio a tratar. La idea consiste en generar un buen aproximador de la función que predice el valor de ejecutar una acción en un estado concreto (Función de valor-acción) la cuál separe en regiones el espacio de estados con

el que está tratando. En una segunda fase, 2SRL aplica técnicas de aprendizaje por refuerzo sobre la representación en forma de regiones generada por el aproximador.

En este proyecto se pretende comprobar la eficacia de la técnica 2SRL en el dominio de la *Robocup-Soccer Keepaway*, un dominio complejo con un espacio de estados continuo. La *Keepaway* es una subtarea del problema del *robosoccer* en el que se enfrentan dos equipos: los *keepers* y los *takers* El objetivo de los *keepers* es mantener la posesión del balón el máximo tiempo posible, mientras que el objetivo de los *takers* es hacer perder la posesión del balón a los *keepers*. Además también se comprobará el rendimiento del algoritmo utilizando diferentes alternativas para la aproximación de la función de valor-acción.

1.2 Objetivos

Parte de los objetivos que se buscan en este trabajo puede de deducirse de lo expuesto en el apartado anterior. A continuación se enumeran de forma más clara los objetivos que persigue este proyecto:

- Comprobar el rendimiento de 2SRL utilizando diferentes técnicas para generar regiones a partir de aproximadores de la función valor-acción en el dominio de la *Robocup-Soccer Keepaway*. Se comprobará el rendimiento de 2SRL con árboles de clasificación y con árboles de regresión.
- Dentro del objetivo de probar 2SRL con árboles de clasificación, se comprobará la influencia de la discretización del espacio de recompensas en los resultados.
- Dentro del objetivo de probar 2SRL con árboles de regresión, se comprobará el rendimiento de conseguido utilizando árboles de regresión que contienen un valor numérico en cada hoja y árboles de regresión que contienen un modelo de regresión lineal cada hoja.
- Probar la influencia en los resultados de utilizar un solo aproximador o utilizar tantos aproximadores como acciones con el fin de aproximar la función de valor-acción. Esta prueba es posible porque el dominio de la *Keepaway* presenta un número reducido de acciones posibles.
- Probar el rendimiento conseguido utilizando como algoritmo de aprendizaje por refuerzo *Q-Learning* y Q(λ).
- Comparar los resultados obtenidos en este proyecto con otras técnicas de generalización que hayan sido validadas previamente en el dominio de la *Robocup-Soccer Keepaway.*

1.3 Visión general del documento

Este documento se estructura de la siguiente forma. El capítulo 2 expone las bases teóricas en las que se sustentan los experimentos realizados y las herramientas utilizadas. Se abordan las técnicas de aprendizaje automático y aprendizaje por refuerzo, y en concreto 2SRL. También se describe el dominio de la *Keepaway*. Por último, en este mismo capítulo se da un enfoque del aprendizaje por refuerzo orientado a la aplicación directa en el dominio de la *Keepaway*, incluyendo referencias a trabajos previos en donde ya se ha aplicado aprendizaje por refuerzo en este dominio.

En el capítulo 3 se expone el diseño que permite la implementación del algoritmo 2SRL sobre el entorno de la *Keepaway*. En primer lugar se presenta el enfoque de diseño de la primera fase del algoritmo. En una segunda parte se explica el diseño que permite la implementación del entorno que se utiliza para ejecutar la segunda fase del algoritmo.

En el capítulo 4 se plasma la experimentación realizada con las implementaciones del diseño del capítulo 3, utilizando varias configuraciones del algoritmo 2SRL. Se muestran los resultados obtenidos y la evolución comparativa de ambas fases del algoritmo. También se realiza una comparativa de los resultados obtenidos con trabajos anteriores que aplicaron otras técnicas de discretización del espacio de estados en *Keepaway*.

En el capítulo 5 se exponen las conclusiones extraídas de la experimentación presentada en el capítulo 4 y se exponen posibles lineas de investigación futuras.

Por último, los anexos A y B presentan la documentación de la implementación del diseño descrito en el apartado 3. El apartado A trata la implementación de la Primera fase de 2SRL y el partado B trata la implementación de la segunda fase de 2SRL.

2. Estado de la cuestión

En este apartado se detallan las bases teóricas sobre las que se sustentan los experimentos realizados en el presente estudio. En primer lugar se relatan las técnicas y herramientas de aprendizaje automático utilizadas. Después se realiza una descripción más detallada de las técnicas de aprendizaje por refuerzo y en concreto del algoritmo *Two Steps reinforcement learning*. Por último, se presenta el dominio en el que se han aplicado las técnicas citadas: la *Robocup Keepaway.*

2.1 Aprendizaje automático

El Aprendizaje Automático trata de construir sistemas informáticos que optimicen un criterio de rendimiento utilizando datos o experiencia previa. Esa experiencia vendrá dada por una serie de datos disponibles sobre el problema a resolver o el criterio a optimizar estructurada en forma de tuplas o vectores de atributos. Estos vectores de atributos $\{X_1, X_2, ..., X_n\}$ o variables predictoras representan un caso del problema. Si disponemos de N de estas tuplas o vectores tenemos N casos de experiencia sobre el problema, quedando nuestra experiencia definida por $D = \{(x_1, \theta_1), (x_2, \theta_2), ..., (x_N, \theta_N)\}$. Cada uno de los θ_i corresponden al valor de clase real al que pertenece cada tupla, haciendo así una clasificación de las tuplas en M clases donde $\theta_i \in \{c_1, c_2, ..., c_M\}$. Con este enfoque muchos problemas de aprendizaje se reducen a un problema de reconocimiento de patrones.

Dentro del mundo del reconocimiento de patrones hay dos grandes grupos de familias que enfocan de manera distinta el problema de la clasificación. Por un lado está la clasificación no supervisada que enfoca la clasificación como el descubrimiento de las clases del problema. Los objetos únicamente vienen descritos por un vector de características, sin que sepamos a la clase a la que pertenece cada uno de ellos. En problema se centra en el descubrimiento de grupos de objetos que según sus características separan las diferentes clases del problema clasificatorio.

Por otro lado, la clasificación supervisada parte de un conjunto de objetos descritos por un vector de características y la clase a la que pertenecen cada uno de ellos. A este conjunto clasificado se le denomina “conjunto de entrenamiento” o “conjunto de aprendizaje”. Así, basándose en este conjunto, la clasificación supervisada construye un “modelo” o “regla general” que será utilizada para clasificar nuevos objetos no presentes en el “conjunto de entrenamiento”.

A continuación se describen algunas técnicas de aprendizaje automático como los árboles de decisión y regresión, haciendo una pequeña introducción a las técnicas de aprendizaje por refuerzo, basadas en la experiencia obtenida mediante ensayo y error.

2.1.1 Árboles de clasificación y regresión

Los árboles de clasificación, también llamados “árboles de decisión”, es uno de los paradigmas de clasificación más utilizados en el mundo del aprendizaje automático.

Los árboles de clasificación forman parte de los métodos de clasificación supervisada, es decir, tendremos un conjunto de entrenamiento clasificado con el que se construirá un modelo que trate de clasificar nuevos casos no presentes en el conjunto inicial (casos de test). La construcción del árbol de clasificación se realiza mediante un proceso de inducción (Top-Down-Induction-Decision-Trees).

El árbol de clasificación puede ser definido como una función $d(x)$ que relaciona cada patrón de entrada x del espacio de clasificación con una clase del conjunto de posibles valores de clase.

Otra definición más próxima al funcionamiento de los árboles de clasificación es considerar al clasificador como una partición del espacio de clasificación X en M subconjuntos disjuntos $A_1, A_2, ..., A_M$ siendo X la unión de todos ellos. El particionado tiene como criterio de agrupación la clase a la que pertenecen las instancias.

Un árbol de clasificación está compuesto por un nodo raíz, nodos intermedios y nodos hoja. El nodo raíz es el primer nodo del árbol de donde cuelgan los demás nodos. Los nodos intermedios o no terminales se encuentran entre el nodo raíz y las hojas. Corresponden con subdivisiones del árbol. Los nodos hoja son aquellos nodos terminales que no se van a dividir más. Cada nodo hoja se corresponderá con una categoría concreta de la variable de clase. De esta manera, los nodos hoja representan las diferentes particiones en las que se ha dividido el espacio de clasificación.

La figura 1 muestra el aspecto de un árbol de clasificación simple. Tanto los nodos intermedios como el nodo raíz corresponden con una pregunta que se le hace a una de las variables del vector de propiedades, también denominada tupla. Cuando la pregunta se responde de manera binaria (Si/No) el árbol es binario y cada nodo padre solo se dividirá en dos nodos hijos correspondientes a cada una de las respuestas posibles. Así, dado un objeto a clasificar, éste va “cayendo” por un lado u otro de los nodos según responda la pregunta correspondiente. Finalmente, la tupla caerá en un nodo hoja que no tiene asignada una pregunta, sino una clase en la que dicha tupla quedará clasificada.

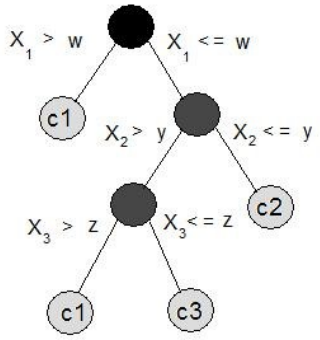

Figura 1: Ejemplo de árbol de clasificación binario.

Como se observa en la figura 1, hay 3 clases (c1, c2 y c3) y 4 nodos hoja. Pueden existir diferentes nodos hoja etiquetados con la misma clase. Al número de nodos hoja que tiene un árbol se le denomina complejidad del árbol y para el caso de la figura la complejidad del árbol sería 4, porque es número de partes en los que ha dividido el espacio de clasificación. Esto es distinto al número de clases. En este caso el árbol ha diferenciado 4 casos distintos a la hora de realizar la clasificación pero dos de los casos corresponden a la misma clase.

Construcción de un árbol de clasificación

A continuación se explica cómo es la fase de entrenamiento que genera como resultado un árbol de clasificación. El proceso de construcción comienza por el nodo raíz. Al principio del proceso, el nodo raíz tiene asociadas todas la tuplas o patrones de entrenamiento. Ahora debemos dividir el conjunto de entrenamiento en un conjunto de particiones que serán los nodos hijos, haciendo que estos subconjuntos dividan de la mejor manera posible el conjunto del nodo padre, intentando que cada hijo solo contenga patrones pertenecientes a una sola clase. Para esta división debemos elegir un atributo discriminante en los patrones que divida la muestra de la manera más homogénea posible. El proceso de subdivisión de los nodos se realiza recursivamente y en el caso extremo, termina cuando los todos los nodos hijos solo contienen muestras pertenecientes a la misma clase. Como se verá más adelante, es deseable parar la subdivisión antes de que se llegue al caso extremo de subdivisión para evitar el fenómeno de “sobreentrenamiento” o “sobreajuste” del modelo.

Entrada: **Conjunto de casos de entrenamiento, E , con sus variables y su clase**
Salida**: Árbol de clasificación (T)**

comienzo
si **Todos los casos en E son de la misma clase C_j**
entonces
  Resultado nodo simple etiquetado como C_j
si_no
  comienzo
    Seleccionar un atributo X_i con valores $x_{i1},...,x_{il}$
    Particionar E en $E_1,...E_l$ de acuerdo con los valores de X_i
    Construir subárboles $T_1,...,T_l$ para $E_1,...E_l$
    El resultado final es un árbol con raíz X_i y subárboles $T_1,...,T_l$
    Las ramas entre X_i y los subárboles están etiquetadas mediante $x_{i1},...,x_{il}$
  fin
fin

Figura 2: Algoritmo general de inducción de árboles de clasificación

Una vez que se decide detener la subdivisión de un nodo, éste será considerado como nodo hoja y se etiquetará con la clase que más esté presente en el conjunto de tuplas de entrenamiento que han "caído" en esa subdivisión. También existe la posibilidad de asignar una probabilidad de pertenecer a una u otra clase presente en la subdivisión. De esta manera una vez construido el árbol, los nuevos ejemplos a catalogar irán "cayendo" desde el nodo raíz hasta los nodos intermedios hasta llegar a un nodo hoja que los clasifique o les asigne una probabilidad de pertenecer a cierta clase. El algoritmo de inducción de árboles de clasificación queda resumido en la figura 2.

Árboles de regresión

Los árboles de regresión se diferencian de los árboles de decisión principalmente en la naturaleza de la variable a predecir. Cuando la variable a predecir es continua se utilizan dos tipos de árboles:

- Árboles de regresión [Quinlan, 1992]: guardan el valor promedio de los valores de las tuplas que clasifican en las hojas.
- Árboles de modelos [Wang y Witten, 1997]: utilizan regresión lineal para predecir los valores de las clases. Cada hoja contiene un modelo lineal que calcula el valor de la clase.

En este tipo de árboles el resultado de la predicción es un número que intenta aproximarse lo mejor posible al valor real que le corresponde a cada ejemplo según sus características. Un ejemplo en el que nos sería útil usar árboles de regresión sería un problema en el que lo que queremos predecir es el valor del crédito que le corresponderá a un cliente según sus características personales, como son el salario, la edad, el número de hijos, etc. (figura 3).

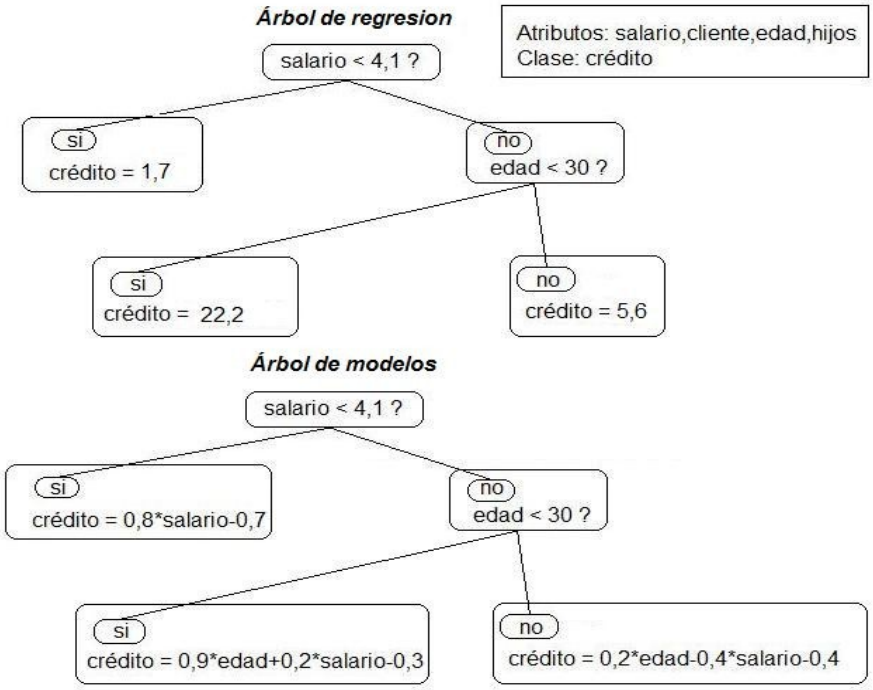

Figura 3: Ejemplo de árboles de regresión.

Árboles "sobreentrenados" y parsimoniosos

Como se comentó anteriormente, no siempre es deseable desarrollar el un árbol de tal modo que todos los nodos hoja sean homogéneos, haciendo que el error de entrenamiento se minimice. En principio, un error menor podría parecer un resultado más deseable. Pero lo que realmente sucede con los árboles que consiguen ajustarse con precisión a las muestras de entrenamiento es que al presentarles nuevos casos, pueden no funcionar bien y clasificar erróneamente. Cuando se produce esta situación se dice que el árbol está "sobreentrenado". Normalmente el "sobreentrenamiento" de un árbol está muy ligado a su complejidad (numero de hojas).

Para evitar el "sobreajuste" conviene construir árboles parsimoniosos con una complejidad suficiente para resolver el problema y que sean capaces de clasificar con éxito nuevos casos no presentes en el entrenamiento (capacidad de generalización).

Los procedimientos que consiguen frenar el desarrollo de los árboles y generar árboles parsimoniosos se denominan procedimientos de *poda.* La

poda puede aplicarse mientras es está desarrollando el árbol (pre-poda) o una vez desarrollado el árbol eliminando subárboles según un criterio determinado (post-poda).

Implementaciones de árboles

- CART [Breiman et al,1984]: Este algoritmo utiliza por defecto el criterio basado en el Gini index para el caso de la clasificación. Cuando se construye un árbol de regresión, los criterios se basan en la mínima suma de desviaciones absolutas o en la minimización de las desviaciones cuadráticas. Para realizar la post-poda realiza una estimación del error, bien mediante un conjunto de datos diferente al que se ha utilizado para construir el árbol, o bien aplicando una metodología de validación cruzada. Introduce el tamaño del árbol en la expresión a minimizar para tener en cuenta la parsimonia del modelo a la hora de podar el árbol.
- ID3 (Iterative Dichotomiser 3) [[Quinlan, 1979], [Quinlan, 1986]]: No realiza ningún proceso de poda y las divisiones se realizan sobre todos los posibles valores de la variable predictora seleccionada. Está limitado al tratamiento de variables discretas. Al utilizar el criterio de la ganancia de información como función de división, tiende a sesgarse hacia variables con un elevado número de categorías diferentes. No soporta ruido.
- C4.5 [Quinlan, 1993]: Sucesor del ID3. Soluciona muchas de las limitaciones de su antecesor. El algoritmo por defecto para las variables discretas genera una rama por cada valor posible de la variable predictora. La versión *C4.5 Release 8* aplica un heurístico a base de unir atributos para determinar una división que mejore la función de división. Con las variables continuas realiza divisiones binarias. Para realizar la post-poda hace una estimación pesimista del error en base a los casos de entrenamiento. La poda no se limita a podar subárboles enteros sino que a veces sustituye un nodo por uno de sus hijos. El criterio de división se basa en la entropía. Una implementación muy difundida de este algoritmo es la conocida como J48 [Frank y Witten, 2005].
- M5 [Quinlan, 1992]: Genera “regression trees” y “model trees” [Wang y Witten, 1997]. Es una variación de CART. En las hojas se calcula un modelo lineal utilizando regresión estándar en función de los valores de los atributos, el cual proporciona un valor numérico (clase predecida). Los modelos lineales se calculan para cada nodo del árbol, considerando sólo los atributos que aparecen en su subárbol como test o en modelos lineales. En cada modelo lineal se eliminan atributos utilizando escalada para reducir el error estimado. Esto normalmente hace que aumente el error residual pero también reduce el factor por el que luego se multiplica. Para la poda cada nodo interno del árbol tiene un modelo simplificado lineal y un modelo subárbol. Se elige aquel que minimice el error. Si es el modelo lineal, el subárbol se queda reducido a ese nodo.

2.1.2 Aprendizaje por refuerzo

El aprendizaje por refuerzo [Kaelbling et al, 1996] es una técnica de aprendizaje que se aplica a agentes donde la interacción con un entorno es muy activa y dinámica. Un ejemplo de agente de este tipo puede ser un robot que tenga que moverse por el espacio sorteando obstáculos, como los robots mandados al espacio para explorar la superficie desconocida de un planeta y tomar muestras para su posterior estudio. Los sistemas de navegación autónomos para vehículos son otro buen ejemplo del tipo de agentes a los que va orientada esta técnica de aprendizaje.

En este marco, el aprendizaje del sistema o agente se realiza mediante un proceso iterativo de ensayo-error durante la interacción con el entorno. Este entorno informa al agente de como de bien está realizando la tarea que se está aprendiendo mediante una señal de refuerzo. Mediante la información que el agente recibe del entorno (estado y refuerzo) el agente puede aprender la tarea especificada cumpliendo un objetivo y optimizando el modo de conseguirlo.

Aprendizaje por refuerzo como un modelo de decisión de Markov

Un agente, ya sea hardware o software, está conectado a un entorno de tal manera que puede recibir ciertas características del mismo y enviar señales de respuesta que actuarán como acciones ejecutadas sobre el entorno. Las características del entorno se conocen como el *estado* s en el que se encuentra el agente en el entorno. Al ejecutar el agente una acción a, se produce un cambio de *estado* a s'. Este nuevo estado es transmitido al agente junto con la señal de refuerzo r correspondiente al par estado-acción anterior que permite al agente saber la utilidad de ejecutar la acción a desde el estado s.

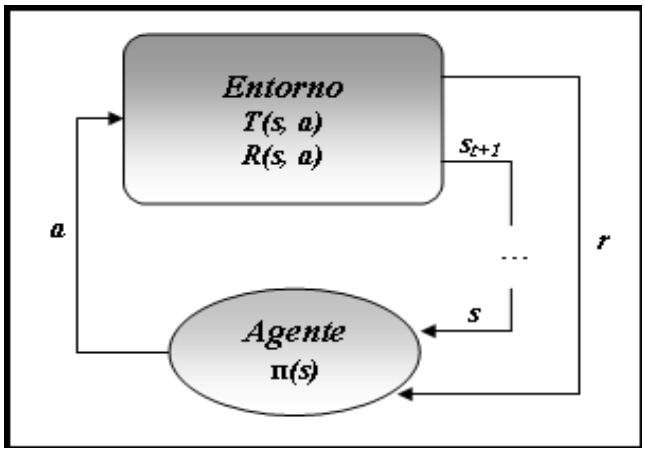

Figura 4: Modelo de aprendizaje por refuerzo

La elección de la acción a ejecutar en cada momento viene dada por una política π. La política devuelve una acción del conjunto de acciones, A,

ejecutables por el agente y una vez ejecutada, el agente transitará a uno de los estados pertenecientes al conjunto de estados posibles, S.

Formalmente, un MDP (*proceso de decisión de Markov*) [Puterman, 1994] se define como una tupla $\langle S, A, \mathcal{T}, \mathcal{R} \rangle$, de forma que:

- S: Conjunto de estados en los que el agente puede estar en cada momento.
- A: Conjunto de acciones que el agente puede ejecutar para de esta forma interactuar con el entorno.
- $\mathcal{T}$: $S \times A \rightarrow P(S)$: donde cada miembro de $P(S)$ es una distribución de probabilidad sobre el conjunto S. $T(s, a, s')$ es la probabilidad de que se transite del estado s a s' ejecutando la acción a.
- $\mathcal{R}$: $S \times A \rightarrow R$: $R(s, a)$ es el refuerzo recibido cuando se ejecutó la acción a en el estado s.

El agente debe encontrar una política, π, que decida la acción a ejecutar en cada momento, considerando que la suma de los refuerzos recibidos a largo plazo debe ser maximizada (o minimizada según el caso). Este requisito de optimización se denomina criterio de optimalidad.

Un ejemplo de criterio de optimalidad es el criterio de horizonte infinito descontado, que tiene como objetivo maximizar la suma de los refuerzos r_k recibidos a lo largo del tiempo:

$$\sum_{k=0}^{\infty} \Upsilon^k r_k$$

La anterior fórmula utiliza un parámetro de descuento, Υ, tal que $0 \leq \Upsilon \leq 1$. Este parámetro reduce la influencia de los refuerzos recibidos en el futuro.

Un problema de aprendizaje por refuerzo se denomina *proceso de decisión de Markov* (MDP) si satisface la propiedad Markov [[Bellman, 1957], [Puterman, 1994]]. La propiedad de Markov se cumple si la probabilidad de que el agente se encuentre en un estado y reciba un determinado refuerzo en el instante $t+1$ sólo depende del estado en el que se encontraba y de la acción ejecutada en el instante t. Esto hace que la toma de decisiones futuras no dependa del camino trazado hasta ese momento, sino sólo del momento actual. Las señales que son capaces de resumir el pasado ocurrido y hacer que para la toma de decisiones sólo sea necesario conocer el estado actual se denominan *markovianas*.

La propiedad de Markov se define matemáticamente de la siguiente manera:

$$Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t\} = Pr\{s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0\}$$

Si el numero de estados y acciones definidas en el modelo son finitos, el MDP se denomina proceso de decisión de Markov finito.

Las funciones $T(s,a,s')$ y $R(s,a)$ se conocen como la dinámica del entorno. Estas funciones resumen las probabilidades de la transiciones de estado y de las funciones de refuerzo, quedando definidas matemáticamente de la siguiente manera:

$$T(s,a,s') = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$

$$\sum_{s_i \in S} T(s,a,s_i) = 1 \quad , \forall\, a \in A \quad y \quad \forall\, s \in S$$

$$R(s,a) = E\{r_{t+1} | s_t = s, a_t = a\}$$

$Pr\{\}$ y $E\{\}$ definen probabilidad y esperanza respectivamente.

Más adelante, en la sección 2.2, se explican con detalle los métodos de resolución existentes para obtener una política óptima y la forma de tratar problemas donde el espacio de estados es muy grande o continuo.

2.1.3 Herramientas de aprendizaje automático

En este apartado nos centraremos en la descripción del software WEKA [Frank y Witten, 2005], utilizado para la generación de árboles en el presente proyecto. Tanto el software como manuales y tutoriales de uso están disponibles en la página oficial http://www.cs.waikato.ac.nz/ml/weka/.

WEKA

Su nombre está compuesto por las siglas de *Waikato Enviornment for Knowledge Analysis*. Es de libre distribución y sus desarrolladores son los investigadores del *Machine Learning Laboratory* de la Universidad de Waikato. El software WEKA es una herramienta de aprendizaje automático y minería de datos escrito en java muy famoso en el ámbito de la investigación que implementa distintas técnicas y algoritmos de minería de datos y modelado predictivo. Posee una interfaz gráfica de fácil manejo pero también permite otros modos de interacción, ya sea mediante comandos o lanzando directamente ejecuciones de los algoritmos desde un intérprete de comandos utilizando la librería JAR de WEKA (java -cp weka.jar).

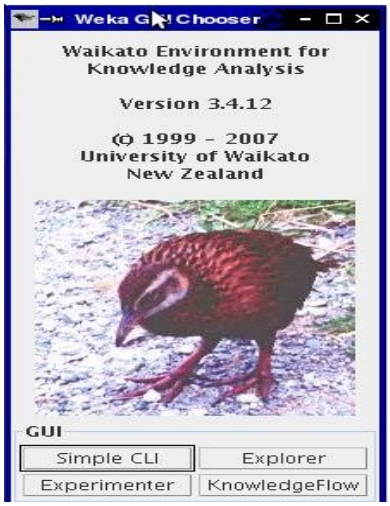

Figura 5: GUI Chooser de WEKA

WEKA soporta varias tareas estándar de minería de datos, especialmente, preprocesamiento de datos, clustering, clasificación, regresión, visualización, y selección. Los datos de entrada para los algoritmos son recogidos en un fichero de texto plano, en el que cada registro de datos está descrito por un número fijo de atributos (normalmente numéricos o nominales, aunque también se soportan otros tipos). También puede obtener los datos mediante acceso a bases de datos vía SQL gracias a la conexión JDBC (Java Database Connectivity) y puede procesar el resultado devuelto por una consulta hecha a la base de datos. En el presente proyecto solo se ha hecho uso de la entrada de datos mediante ficheros ARFF que serán descritos más adelante.

Modos de utilización de WEKA

En el presente apartado se especifican los modos en los que WEKA puede ser utilizado.

- Simple CLI: abreviatura de “Simple Command Line Interface”. Consiste en una consola en la que se puede interactuar con WEKA mediante comandos. También podemos usar un prompt del sistema operativo e invocar los algoritmos de WEKA especificando en el classpath la ubicación de la librería JAR que contiene el paquete WEKA. (*java -cp weka.jar weka.classifiers.trees.J48 -C 0.25 -M 2 -t rweka.arff -d rweka.model*)

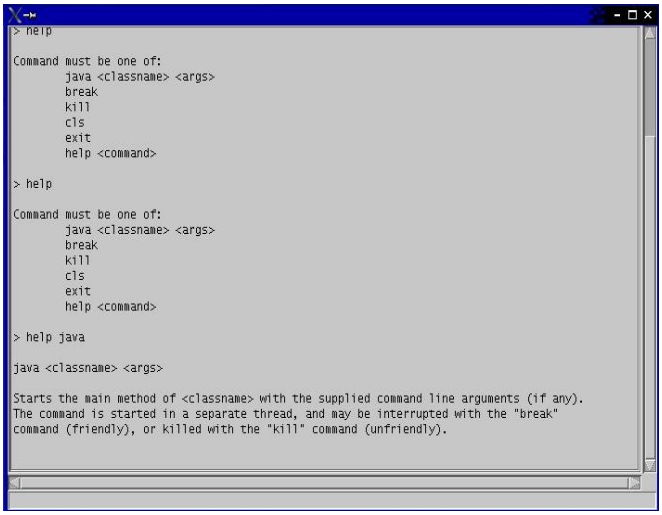

Figura 6: Simple CLI

- Explorer: Consiste en una interfaz gráfica con paneles y pestañas que permiten la carga de datos, la configuración y la ejecución de distintos procedimientos. Dentro de lo que se puede hacer en el modo explorer tenemos las siguientes opciones:

  - Preprocess: Este panel se usa para la carga y transformación de los datos. Podemos cargar los datos desde fichero, desde una URL o desde una base de datos. Una vez cargados los datos tenemos la posibilidad de aplicar filtros y transformaciones sobre ellos.

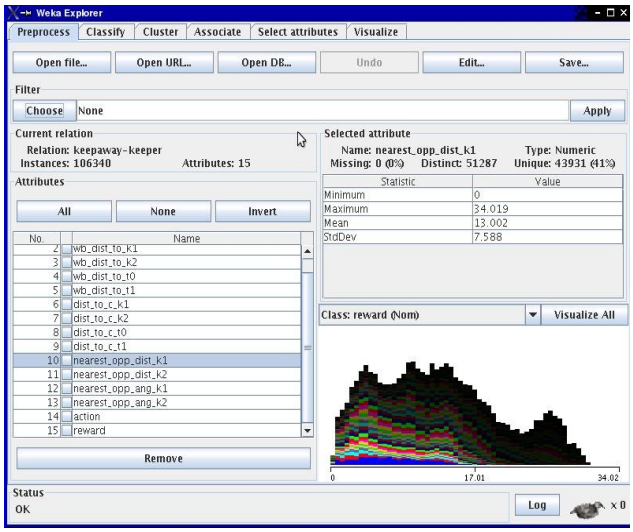

Figura 7: Preprocess

- Classify: Desde aquí se pueden aplicar técnicas de clasificación y regresión a los datos resultantes del preproceso (preprocess). Los clasificadores y modelos resultantes pueden ser evaluados con diferentes técnicas que se configuran desde este panel. Además los parámetros de los algoritmos empleados también son configurables. Hay una gran variedad de algoritmos que se pueden aplicar. Alguno de los algoritmos disponibles son (versión 3.4.12) NaiveBayes, J48 (C4.5) y M5.

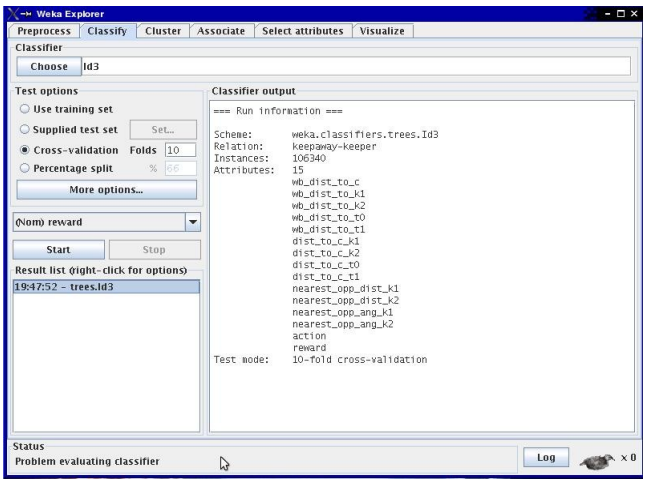

Figura 8: Classify

- Cluster: Desde aquí se pueden aplicar las técnicas de *clustering* o agrupamiento sobre el conjunto de datos cargado. Una de estas técnicas es el algoritmo K-medias.

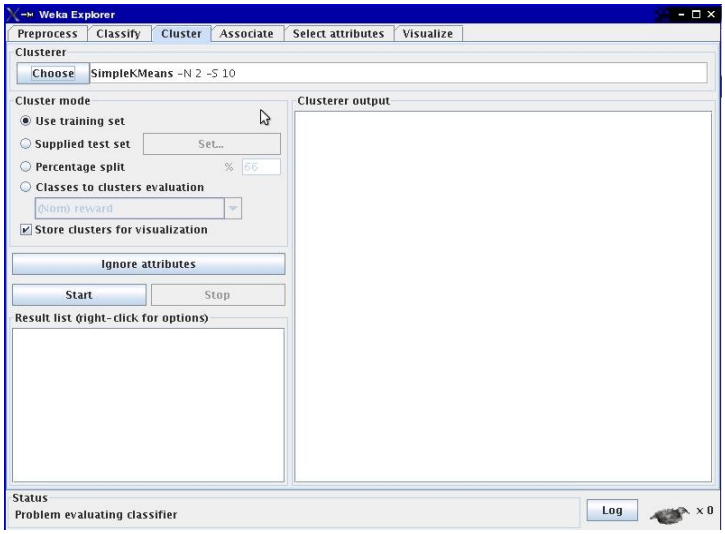

Figura 9: Cluster

- Associate: Proporciona acceso a las reglas de asociación aprendidas que intentan identificar todas las interrelaciones importantes entre los atributos de los datos.

- Select attributes: Proporciona algoritmos para identificar los atributos más predictivos en un conjunto de datos.

- Visualize: muestra una matriz de puntos dispersos donde cada punto individual puede seleccionarse y agrandarse para ser analizados en detalle usando varios operadores de selección.

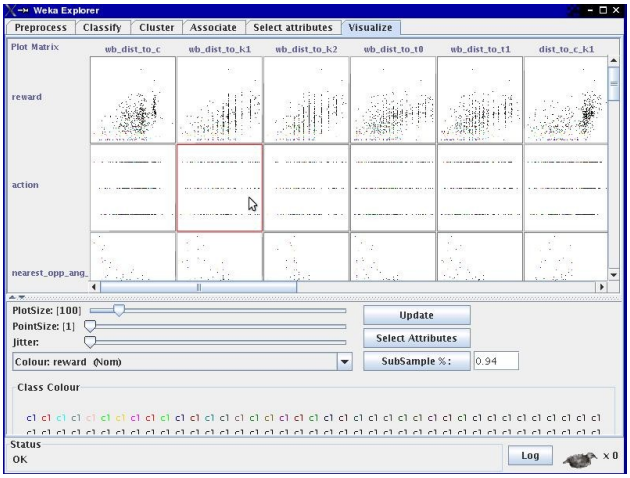

Figura 10: Visualize

- Knowledge Flow: soporta esencialmente las mismas funciones que el Explorer pero con una interfaz "drag and drop". Permite encadenar procedimientos.

- Experimenter: permite lanzar una batería de experimentos en modo *batch*. Podemos configurar los distintos datos a cargar, los distintos algoritmos que se aplicarán sobre esos datos y las técnicas de evaluación con las que se testarán los resultados obtenidos.

Formato de salida de las técnicas utilizadas

Como veremos más adelante, es muy importante conocer exactamente el tipo de salida que nos proporcionan los algoritmos de WEKA para poder trasladar estos resultados a entornos que nos permitan probar su eficacia real. Si, como es el caso, necesitamos implementar los modelos obtenidos por los algoritmos especificando de programación y además queremos hacer numerosas pruebas en el menor tiempo posible, será necesario implementar un "parser" o traductor que nos facilite la tarea de transcripción de un lenguaje formal o pseudocódigo a un lenguaje de programación concreto. En este apartado se analiza la salida obtenida de las principales técnicas utilizadas en este proyecto.

- weka.classifiers.trees.J48

  Implementación de C4.5 [Quinlan, 1993]. Se utiliza cuando el atributo de clase se representa con valores discretos. Los valores del resto de los atributos pueden ser continuos o discretos. Su salida es un árbol construido a partir de las muestras disponibles. La división de los nodos es binaria cuando el atributo por el que se divide el nodo es continuo y *n*-aria cuando el atributo por el que se divide el nodo es discreto o enumerado. Las hojas están etiquetadas con la clase que clasifican a las tuplas que lleguen a ellas. En la figura 11 se muestra la salida de un ejemplo sencillo en modo texto y en la figura 12 su visualización en el interfaz gráfico.

```
J48 pruned tree
------------------

outlook = sunny
|   humidity <= 75: yes (2.0)
|   humidity > 75: no (3.0)
outlook = overcast: yes (4.0)
outlook = rainy
|   windy = TRUE: no (2.0)
|   windy = FALSE: yes (3.0)
```

Figura 11: salida modo texto de un ejemplo de J48.

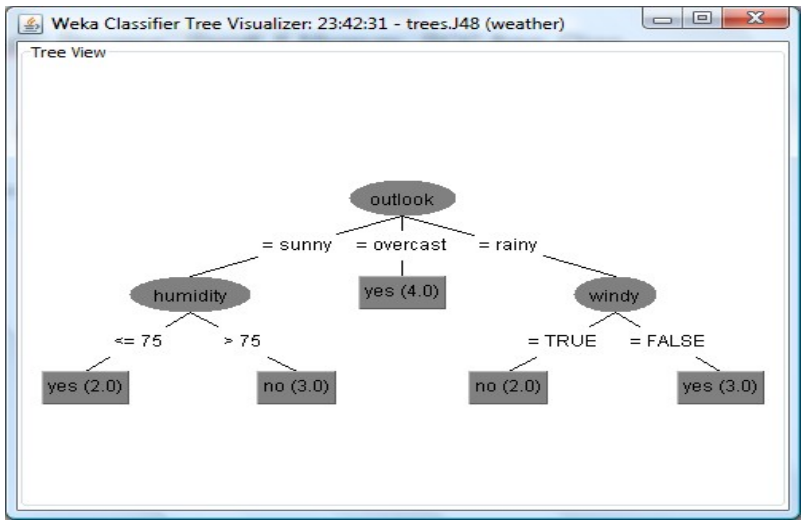

Figura 12: Visualización árbol J48 en WEKA.

- weka.classifiers.trees.M5P

Implementa los algoritmos para generar M5 “model trees” [Wang y Witten, 1997] y “rules” [Quinlan, 1992]. El algoritmo original M5 fue inventado por R. Quinlan, y Yong Wang realizó mejoras sobre él. Se utiliza cuando el atributo de clase se representa con valores continuos. Los valores del resto de los atributos pueden ser continuos o discretos. Su salida es un árbol construido a partir de las muestras disponibles. La división de los nodos es binaria cuando el atributo por el que se divide el nodo es continuo y *n*-aria cuando el atributo por el que se divide el nodo es discreto o enumerado. Las hojas, dependiendo de si se usa la opción -R (regression) o no, contienen la media de los valores de clase correspondientes a las instancias clasificadas en esa hoja o un modelo de regresión lineal que calcula el valor de clase. En la figura 13 se muestra la salida de un ejemplo sencillo en modo texto.

```
M5 pruned regression tree:
(using smoothed linear models)

CACH <= 8.5 :
|   MMAX <= 6100 : LM1 (75/3.056%)
|   MMAX >  6100 :
|   |   MYCT <= 83.5 :
|   |   |   MMAX <= 10000 : LM2 (8/2.545%)
|   |   |   MMAX >  10000 : LM3 (3/2.137%)
|   |   MYCT >  83.5 : LM4 (22/3.81%)
CACH >  8.5 :
|   CHMIN <= 7 :
|   |   MYCT <= 95 : LM5 (7/7.521%)
|   |   MYCT >  95 : LM6 (18/5.501%)
|   CHMIN >  7 : LM7 (8/25.484%)
```

Figura 13: salida modo texto de un ejemplo de M5.

2.2 Aprendizaje por refuerzo

En el apartado 2.1.2 se explicó cuál es la filosofía de esta técnica y cómo definir un problema de aprendizaje por refuerzo como un MDP. En este apartado profundizaremos en los procedimientos para obtener una política óptima y la manera de aplicar el aprendizaje por refuerzo en dominios complejos. Se puede obtener más información sobre este tema en [Sierra et al, 2006].

2.2.1 Funciones de valor

Las funciones de valor son utilizadas por un gran número de algoritmos de aprendizaje por refuerzo para aprender la política π. Estas funciones estiman como de bueno es para un agente estar en un estado determinado o como de bueno es usar una acción desde un determinado estado utilizando una política π. Se detona con $V^{\pi}(s)$ al refuerzo que se espera obtener si comenzamos a guiarnos por la política π desde el estado s hasta el infinito. La definición formal para un MDP es la siguiente:

$$V^{\pi}(s) = E_{\pi}\{\sum_{k=0}^{\infty} \Upsilon^{k} r_{t+k+1} | s_t = s\}$$

$E_{\pi}\{\}$ denota el valor esperado cuando el agente sigue la política π. A V^{π} se le denomina función de valor-estado. En la ecuación se introduce el factor de descuento del horizonte infinito Υ definido en la sección 2.1.2.

También podemos definir $Q^{\pi}(s,a)$ como el refuerzo esperado al ejecutar una acción a desde un estado s siguiendo una política π. En este caso a Q^{π} se le denomina función de valor-acción. La definición formal es:

$$Q^{\pi}(s,a) = E_{\pi}\{R_t | s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \Upsilon^{k} r_{t+k+1} | s_t = s, a_t = a\}$$

Dentro de todas las posibles políticas existen siempre una o más políticas que son mejores o iguales al resto. A las políticas que cumplen esta condición se les denomina políticas óptimas π^{*}. Las políticas óptimas comparten una única función de valor-estado que maximiza el valor para cualquier política. Esta función se denomina función de valor-estado óptima $V^{*}(s)$.

$$V^{*}(s) = \max_{\pi} V^{\pi}(s), \forall s \in S$$

También se puede definir la función de valor-acción óptima $Q^{*}(s,a)$, la cuál también será única. Esta función maximiza el valor de cualquier par estado-acción para cualquier política.

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a), \forall s \in S, \forall a \in A$$

Las políticas óptimas pueden derivarse fácilmente de las funciones de valor óptimas. Si utilizamos la función valor-acción la mejor acción a ejecutar desde el estado s será la que maximice el valor de dicha función. Si existen varias acciones que maximizan el valor de $Q^*(s,a)$ podremos encontrar varias políticas óptimas.

$$\pi^*(s) = arg_a\, máx\, Q^*(s,a)$$

2.2.2 Métodos de resolución

Existen diversos métodos de resolución para los problemas de aprendizaje por refuerzo. Estos métodos se basan en el mantenimiento de las *funciones de valor* para estructurar el conocimiento. Obteniendo los valores óptimos de estas funciones se obtiene una representación óptima de la política a seguir. Estas funciones óptimas cumplen las ecuaciones de optimalidad de Bellman [Bellman, 1957] que formulan matemáticamente el siguiente principio de optimalidad: *Una política óptima tiene la propiedad de que cualesquiera que sean el estado inicial y la primera decisión tomada, el resto de decisiones debe constituir una política óptima respecto al estado resultante de la primera decisión.*

Las técnicas de *programación dinámica* se utilizan cuando se posee un conocimiento completo de un modelo perfecto del entorno como un Proceso de Decisión de Markov. Es decir, se conocen los conjuntos de estados y acciones y las funciones de transición de estados y de refuerzo.

Raras veces tenemos un conocimiento completo del modelo. En la mayor parte de los casos disponemos de un conocimiento parcial del problema. Ante esta situación existe la posibilidad de aprender primero la dinámica del problema mediante exploración para después emplear las técnicas de programación dinámica. Estos métodos son conocidos como *Métodos basados en el modelo*.

Por último, los *métodos libres de modelo* basan el cálculo las funciones de valor en la experiencia obtenida durante la interacción con el entorno, sin necesidad de tener un conocimiento explícito de la dinámica del mismo. A continuación nos centraremos en este tipo de métodos, que son los que mejor se pueden aplicar a problemas reales, en los cuales normalmente no tenemos gran conocimiento del modelo.

Métodos libres de modelo

En este tipo de métodos al igual que los métodos basados en el modelo se asume un desconocimiento total de la dinámica del entorno. Sin embargo, los métodos libres de modelo se basan solo en la experiencia que obtienen de interactuar con el entorno para calcular las funciones de valor, aprendiendo de manera implícita la dinámica desconocida. La forma de interacción del agente en este tipo de técnicas puede ser diversa, dando lugar a diferentes métodos:

- *Métodos de Monte Carlo*. En la interacción del agente con el entorno se pueden extraer secuencias completas de comportamiento desde una situación de partida hasta una situación final. Estas secuencias completas de comportamiento no serían más que la secuencia de recompensas observadas por el agente mientras interactuaba con el entorno.

- *Métodos de diferencia temporal (TD Methods)*. Estos métodos tratan de utilizar esta experiencia paso a paso, o lo que es lo mismo, por cada acción realizada por el agente. La actualización de los métodos de *TD* la hacen utilizando únicamente la siguiente recompensa.

En cualquier caso, todos estos métodos se basan en las funciones de valor definidas anteriormente para representar las políticas de acción. Además dependiendo del uso que se haga de la política que se está aprendiendo distinguimos entre métodos "*on-policy"* que durante el aprendizaje utilizan la política de acción que están aprendiendo y los métodos "*off-policy"* donde la política de interacción es independiente de la política que se esta aprendiendo.

Q-Learning

El algoritmo *Q-Learning* [Watkins, 1989] es uno de los desarrollos más importantes en aprendizaje por refuerzo de un algoritmo *off-policy*. Pertenece a los *métodos libres de modelo* y enfoca el aprendizaje a aproximar la función estado-acción. Este algoritmo aprende a partir de la experiencia generada durante la exploración del entorno. Esta experiencia se representa mediante las denominadas tuplas de experiencia $\langle s, a, s', r \rangle$, donde s es el estado actual, a es la acción ejecutada, s' es el estado al que se ha llegado, y r es el refuerzo recibido. Este algoritmo se puede aplicar directamente procesando *on-line* los refuerzos obtenidos durante la interacción o pueden enfocarse métodos *off-line* que utilizan tuplas de experiencia recogidas a lo largo de una ejecución anterior del agente con el entorno. En la función de actualización de la tabla Q se incluye un parámetro de enfriamiento α, que define la importancia que se da a las nuevas actualizaciones con respecto a valores anteriores. Este parámetro se utiliza cuando el dominio es estocástico, es decir, cuando el resultado de ejecutar una acción sobre un mismo estado puede generar transiciones de estado o refuerzos distintos. El parámetro α debe ser un valor entre 0 y 1. Cuando $\alpha = 1$ se obtiene la función de actualización para entornos deterministas. El algoritmo completo queda resumido en la figura 14.

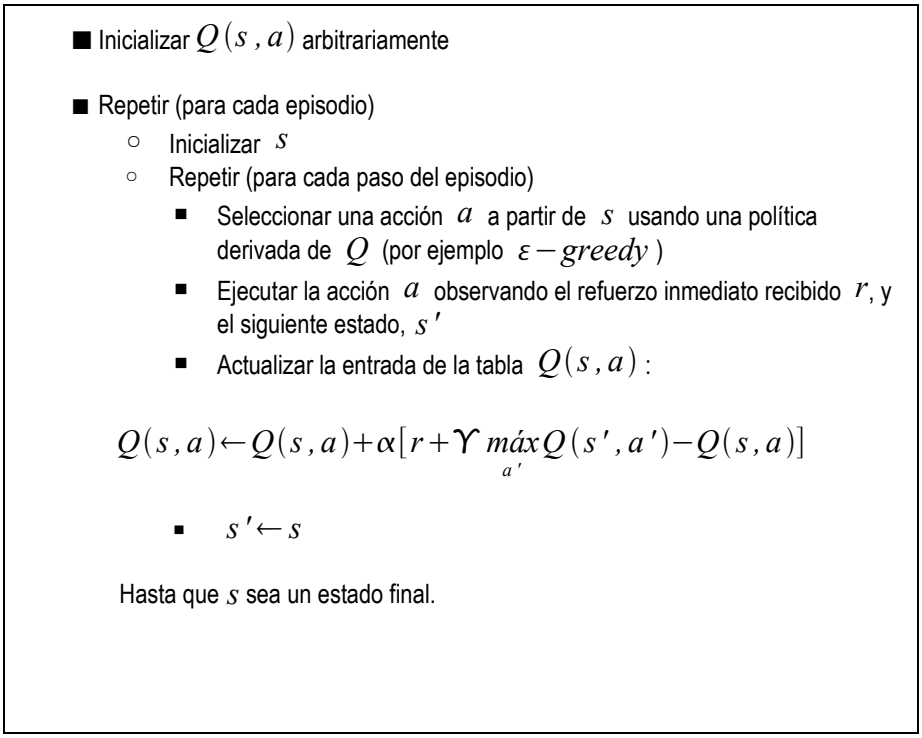

Figura 14: Algoritmo Q-Learning

Métodos TD(λ) y trazas de elegibilidad

A continuación se discuten los métodos $TD(\lambda)$ que son métodos intermedios entre los métodos *Monte Carlo* que utilizan secuencias completas de recompensas entre el estado inicial y el final, y los métodos $TD(0)$ que solo utilizan la siguiente recompensa en la interacción con el entorno.

Tanto *Monte Carlo* como $TD(0)$ utilizan una actualización en la que el nuevo valor estimado para la función de valor de un estado se obtiene haciendo que el valor estimado calculado con anterioridad tienda a un objetivo en la proporción de una tasa de aprendizaje. En *Monte Carlo* se utiliza como objetivo una secuencia completa como se muestra a continuación:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T$$

donde T es el tiempo del último paso del episodio.

En cambio en $TD(0)$ se utiliza solo la primera recompensa más el valor descontado del próximo estado:

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$$

el sumando $\Upsilon V_t(s_{t+1})$ sustituye a la secuencia
$\Upsilon r_{t+2}+\Upsilon^2 r_{t+3}+...+\Upsilon^{T-t-1} r_T$ de *Monte Carlo*.

El objetivo de $TD(0)$ se dice que es un objetivo truncado después de un paso de tiempo *(n-step target)*. El objetivo de dos pasos *(two-step target)* se definiría como $R_t^2=r_{t+1}+\Upsilon r_{t+2}+\Upsilon^2 V_t(s_{t+2})$ y para n pasos:

$$R_t^n=r_{t+1}+\Upsilon r_{t+2}+\Upsilon^2 r_{t+3}+...+\Upsilon^{n-1} r_{t+n}+\Upsilon^n V_t(s_{t+n})$$

El algoritmo $TD(\lambda)$ puede ser tratado desde dos enfoques:

- *Enfoque hacia delante*. Se puede entender como una actualización de la estimación de la función de valor utilizando como objetivo un valor λ que es proporcional a la suma de todos los retornos de n pasos, cada uno de ellos ponderado por λ^{n-1} donde $0\leq\lambda\leq 1$. El objetivo de $TD(\lambda)$ queda expresado matemáticamente de la siguiente manera:

$$R_t^\lambda=(1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}R_t^n$$

  y la actualización de la función de valor se hace conforme a la siguiente ecuación:

$$V_t=V_{t-1}(s_t)+\alpha[R_t^\lambda-V_{t-1}(s_t)]$$

  Cuanto más cerca está λ de 1 más se acerca al comportamiento de Monte Carlo. En cambio, si el valor de alfa se acerca a cero el comportamiento será más cercano a $TD(0)$.

  El gran inconveniente de este enfoque es que no puede implementarse debido a su no causalidad, es decir, en cada paso de tiempo utiliza información de lo que va a suceder en pasos de tiempo futuros.

- *Enfoque hacia atrá*s. Hace posible poder implementar $TD(\lambda)$ gracias al uso de una variable de memoria adicional asociada a cada estado llamada traza de elegibilidad. Cada traza de elegibilidad nos da información de si el estado fue visitado recientemente. En cada paso de tiempo estas trazas se actualizan según la siguiente ecuación:

$$e_t(s,a)=\begin{cases}\Upsilon\lambda e_{t-1}(s,a) & si\ s\neq s_t\\ \Upsilon\lambda e_{t-1}(s,a)+1 & si\ s=s_t\end{cases}$$

  En el caso de las trazas reemplazantes, se sustituye su valor por 1 como se puede observar en la siguiente ecuación:

$$e_t(s,a) = \begin{cases} \gamma \lambda e_{t-1}(s,a) & si\ s \neq s_t \\ 1 & si\ s = s_t \end{cases}$$

El factor e_t es la traza de elegibilidad del estado s en el tiempo t, γ es el factor de descuento y λ es misma variable vista en el enfoque hacia delante y que ahora sirve como parámetro de decaimiento de la traza. Si un estado no es visitado en el paso de tiempo t su traza se decrementa exponencialmente según $\gamma\lambda$, mientras que si es visitado su traza se incrementa en 1 para trazas acumulativas o se establece a 1 en el caso de trazas remplazantes.

La evaluación de la política según $TD(\lambda)$ produce, por tanto, actualizaciones en los valores de la función de valor correspondiente a cada estado dependiendo de cómo de reciente han sido visitados visitados de acuerdo a sus trazas de elegibilidad según la ecuación de actualización:

$$V_t(s) = V_{t-1}(s) + \alpha \delta_t e_t(s), \forall s \in S$$

donde

$$\delta = r + \gamma V(s') - V(s)$$

A continuación se introduce el algoritmo *Q-learning* con trazas de elegibilidad. En concreto la versión del algoritmo representado en la figura 15 se conoce como Watkins's $Q(\lambda)$.

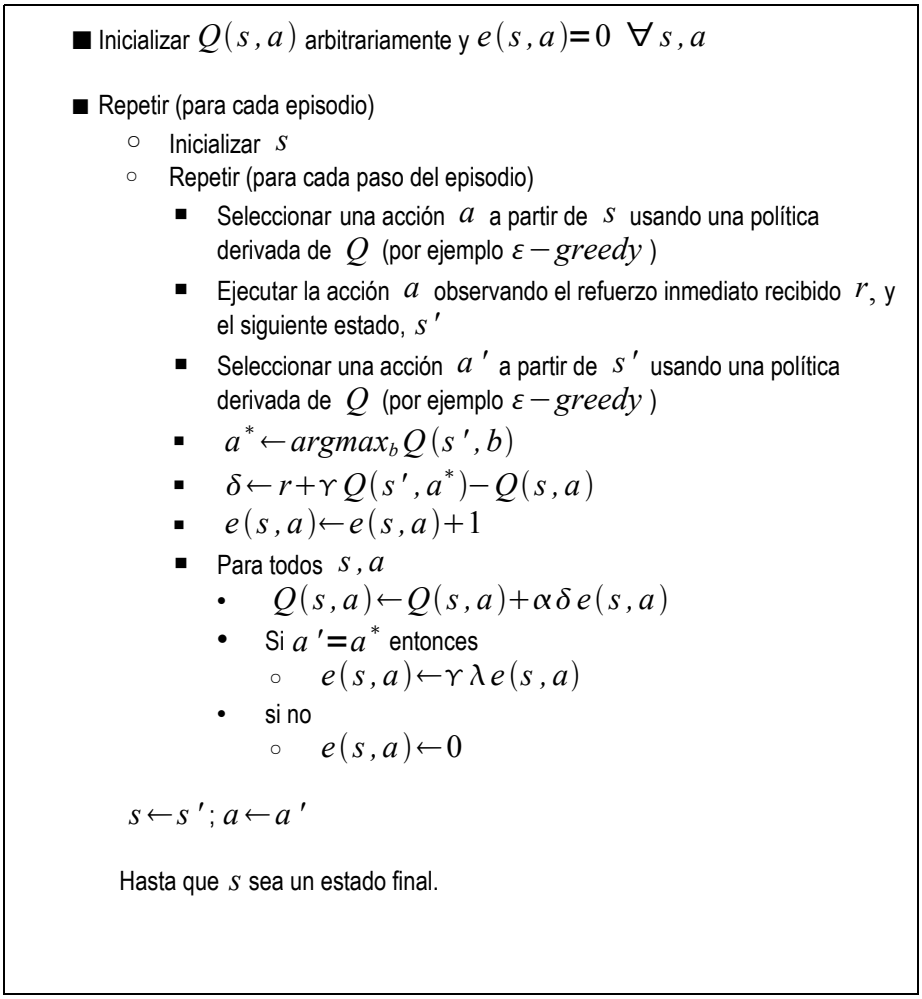

Figura 15: Watkins's Q(λ).

Exploración y explotación

En aprendizaje por refuerzo la exploración es entendida por la ejecución de acciones de forma más o menos aleatoria, de forma que se visiten nuevos estados y por lo tanto nuevas alternativas de solución. En cambio, por explotación se entiende el utilizar la información obtenida en el aprendizaje para maximizar el beneficio.

Para establecer un equilibrio entre exploración y explotación se pueden seguir diversas estrategias. En los extremos de estas estrategias están por un lado las estrategias que siguen un comportamiento totalmente aleatorio y las estrategias denominadas totalmente avariciosas que seleccionan siempre la acción que se supone mejor en un momento determinado. Entre estas dos estrategias se sitúan un amplio abanico de técnicas. Entre estas técnicas destaca $\varepsilon - greedy$ [Watkins, 1989] donde el parámetro ε define la probabilidad de ejecutar la mejor acción posible en un momento dado y $1-\varepsilon$ define la probabilidad de ejecutar acciones aleatorias. Uno de los inconvenientes de la estrategia $\varepsilon - greedy$ consiste en que cuando se decide explorar (seleccionar una acción aleatoria) se elige equiprobablemente entre todas las acciones disponibles. Esto significa que es tan probable seleccionar la mejor acción como la peor. Para solucionar estos inconvenientes surgen otro tipo de técnicas conocidas como *softmax* que asignan probabilidades a cada acción en función del valor que tienen para Q.

2.2.3 Generalización en aprendizaje por refuerzo

En la mayoría de los dominios reales los estados se representan mediante un conjunto de atributos que pueden ser continuos. Además las acciones también pueden estar definidas de forma continua. Cuando sucede ésto, no es posible realizar una representación tabular (figura 16) de las funciones de valor que normalmente utilizan las técnicas de aprendizaje por refuerzo vistas anteriormente. Por otra parte, si el conjunto de estados y acciones es discreto pero demasiado grande el problema de actualización de la tabla Q e incluso el almacenamiento de la misma se vuelve intratable.

Las técnicas de generalización que se exponen a continuación proponen soluciones a los problemas planteados.

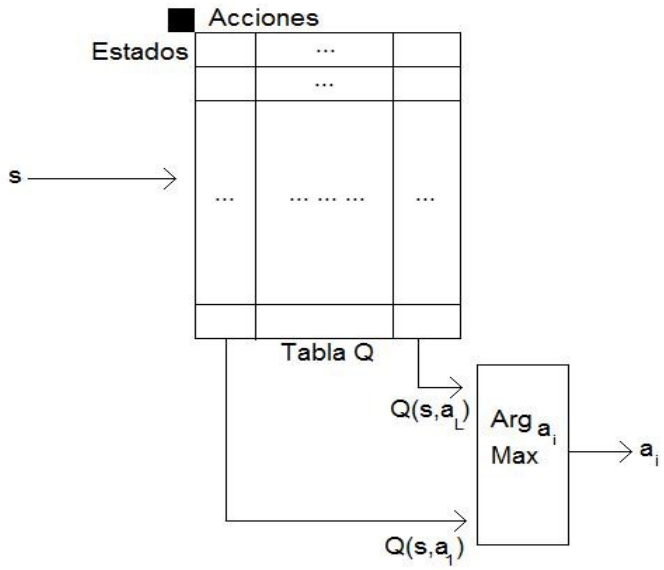

Figura 16: Representación tabular de la tabla Q

Discretización

Cuando tenemos un espacio de estados continuo e infinito o demasiado grande para ser tratado por los métodos de resolución tradicionales, necesitamos buscar una representación alternativa del espacio de estados que resuma las características más relevantes para la tarea que se está llevando a cabo. En cierto modo es como introducir conocimiento *a priori* en el sistema de aprendizaje.

Esta representación en características deriva en como conseguir espacios de estados finitos de un tamaño que pueda ser tratado. La figura 17 muestra como utilizar una tabla para aproximar la función Q utilizando una discretización del espacio de estados original.

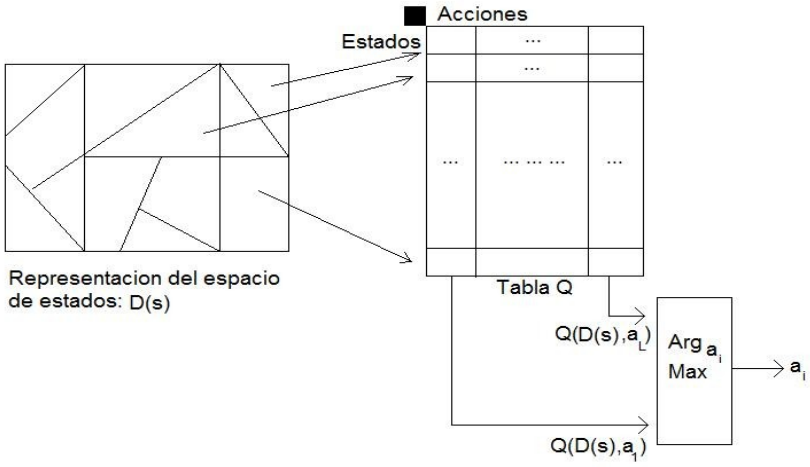

Figura 17: Representación de Q(s,a) basada en discretización

Algunos de los modelos de discretización más importantes son:

- El vecino más cercano. La regla del vecino más cercano [[Duda y Hart, 1973], [Gersho y Gray, 1992]] proporciona una forma sencilla de dividir un espacio de estados en regiones. Cada punto del espacio de estados continuo puede aproximarse mediante una medida de distancia a uno de entre un conjunto de n prototipos, definiendo de forma sencilla las distintas regiones en las que se divide el espacio de estados. Por ejemplo podemos utilizar técnicas como LVQ [Kohonen, 1989] o cuantificación vectorial [Gersho y Gray, 1992].

- Arboles de decisión. Los árboles de decisión [Quinlan, 1986] también han sido utilizados ampliamente en la literatura de aprendizaje por refuerzo. No obstante, los árboles pueden ser construidos o utilizados de dos formas distintas. Una primera aproximación consiste en utilizarlos como aproximadores de funciones puros, incluidos como tales en algoritmos como *Smooth Value Iteration* [Boyan y Moore, 1995] o *Iterative Smooth Q-learning* [Fernandez, 2002]. Otra posibilidad es que la construcción del árbol esté adaptada al problema de aproximar la función de valor. En cualquier caso, los árboles de decisión pueden ser entendidos como una nueva forma de discretizar el espacio de estados, ya que cada hoja puede representar una de estas regiones.

- Otros. Existen otros métodos también muy extendidos como codificación gruesa (coarse coding) [Hinton, 1984], codificación en teja (tile coding) [[Albus, 1971], [Albus, 1981]] y funciones de base radial (RBFs) [Broomhead y Lowe, 1988].

Aproximación de funciones

Mediante la aproximación de funciones se trata de aproximar la función de valor-estado o la de valor-acción con una función de un conjunto de parámetros representados por un vector, en vez de con una tabla como se había hecho en apartados anteriores. De esta manera la función de valor podría implementarse mediante una red de neuronas o un árbol de decisión que recibiría como entrada un vector que representa el estado actual y devolvería la estimación del valor para ese estado. Si entrenamos la red con tuplas recogidas durante la experiencia del agente con el entorno, los problemas de aprendizaje por refuerzo se pueden reducir a simples problemas de aproximación de funciones en aprendizaje supervisado. Sin embargo, la mayoría de estas técnicas presuponen un conjunto de entrenamiento estático sobre el que se realizan una o varias pasadas, mientras que en aprendizaje por refuerzo se dispone de información generada de forma iterativa con un entorno dinámico. Por tanto, se necesitan métodos que funcionen bien con información adquirida de forma incremental.

Cuando intentamos aproximar la función de valor-acción, como ocurre en los métodos libres de modelo, tenemos que incluir un parámetro más, que es la acción. Existen dos enfoques al respecto:

- Podemos incluir la acción como un parámetro más de la función, de forma que el tipo de actualizaciones que se producen son del tipo $s, a \rightarrow q$.

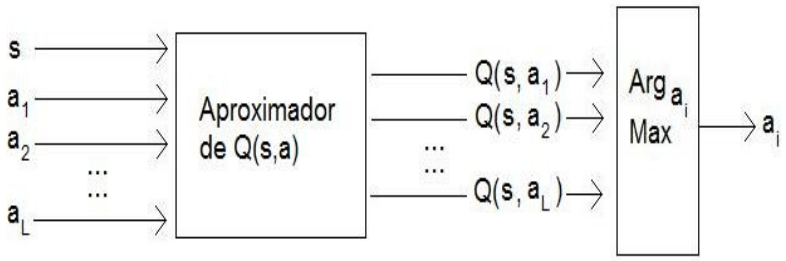

Figura 18: Aproximación de la función Q con 1 aproximador.

- Si el conjunto de acciones es reducido se pueden generar tantos aproximadores como posibles acciones haya. De este modo si tenemos L acciones posibles contaremos con $Q_{a_i}(s), i = 1,2,...,L$ aproximadores distintos.

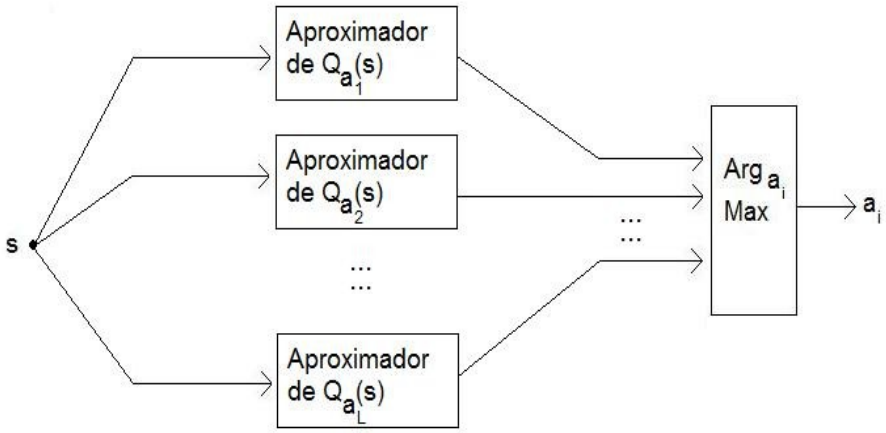

Figura 19: Aproximación de la función Q con L aproximadores.

Para conseguir una buena aproximación de las funciones de valor se utilizan métodos que minimicen el error cometido por los aproximadores. Los métodos de descenso de gradiente tienen como objetivo la reducción del error cuadrático medio pero implican, por una parte, definir la arquitectura de la red y el método de aprendizaje de los pesos, si utilizamos una red de neuronas como aproximador y, por otra parte, conocer *a priori* la función de valor que se está aproximando. Estas premisas que en los problemas de aprendizaje supervisado se cumplen siempre, no se cumplen para el aprendizaje por refuerzo, ya que la función de valor es precisamente lo que se está calculando. Como alternativa, el algoritmo *Smooth Value Iteration* [Boyan y Moore, 1995] plantea una versión de *Value Iteration*, en el que se sustituye la tabla de todos los estados por un aproximador de la función de valor. El aproximador se aprende de forma iterativa, generando tantas aproximaciones $\hat{V}^{iter}$ como iteraciones del algoritmo. En cada una de las iteraciones, el aproximador $\hat{V}^{iter}$ se recalcula utilizando como datos de entrenamiento los estados de ejemplo, cada uno de ellos etiquetados con el nuevo coste, $v^{iter}[i]$, calculado a partir de la aproximación de la función de valor aprendida en la iteración anterior. El algoritmo se plantea como una aproximación de coste a meta, donde se intenta minimizar el coste en alcanzar la meta. La función de valor se va aprendiendo propagando los costes desde la meta (coste 0) hacia el resto del entorno, aprendiendo en cada iteración el coste a meta de los estados situados a una distancia de una acción de los estados para los cuales se había aprendido el coste en la iteración anterior. Este procedimiento queda resumido en la figura 20.

- Dados
  - Una colección de estados $\hat{S}=\{x_1,\ldots,x_N\}$, obtenidos del espacio de estados total S y una zona de meta $G\subset S$
  - Un conjunto de acciones finito A
  - Una función de transición de estados determinista $T:S\times A\to S$
  - Una función de coste $Coste:S\times A\to R$
  - Un aproximador de la función de valor, $\hat{V}^0:S\to R$
- $iter=0$
- $v^0[i]=0,\ \forall i=1,\ldots,N$
- Repetir
  - Entrenar $\hat{V}^{iter}$ para aproximar el conjunto de entrenamiento formado por: $\langle x_1, v^{iter}[1]\rangle,\ldots,\langle x_N, v^{iter}[N]\rangle$
  - $iter=iter+1$
  - Para i=1 hasta N:

$$v^{iter}[i]=\left\{\begin{array}{ll} 0 & si\ x_i\in G \\ mín_{a\in A}\{Coste(x_i,a)+\hat{V}^{iter-1}(\tau(x_i,a))\} & en\ otro\ caso \end{array}\right\}$$

Hasta que el vector v no cambie

- Devolver $\hat{V}^{iter-1}$

Figura 20: Algoritmo Smooth Value Iteration.

2.2.4 Two Steps Reinforcement Learning

Two steps reinforcement learning (2SRL) [Fernandez y Borrajo, 2008] es una técnica que combina el refinamiento de un estimador de la función Q, que puede ser usado para obtener una discretización del espacio de estados, con los algoritmos clásicos de aprendizaje por refuerzo como *Q-Learning*. El método se basa en encontrar discretizaciones del espacio de estados adaptadas a la función de valor que está siendo aprendida. En este caso se aprende la función de valor-acción en vez de la función de valor-estado aprendida en programación dinámica.

La técnica se divide en dos fases de aprendizaje. La primera fase consiste en una versión libre de modelo del algoritmo *Smooth Value Iteration* [Boyan y Moore, 1995] visto en el apartado anterior, denominado *Iterative Smooth Q-Learning (ISQL)* [Fernandez, 2002]. Este algoritmo, el cual ejecuta un aprendizaje iterativo y supervisado de la función Q, puede ser usado para obtener una discretización del espacio de estados. Esta nueva discretización es usada en la segunda fase de aprendizaje, llamada Multiple discretization *Q-Learning* (MDQL) para obtener una política mejorada. Juntas, las fases ISQL y MDQL componen el algoritmo *Two Steps Reinforcement Learning* (2SRL).

Fase 1: ISQL

El algoritmo *Iterative Smooth Q-Learning* es una versión libre de modelo del *Smooth Value Iteration*, en el que se trata de aproximar Q(s,a) utilizando un aproximador por acción y la función de actualización del algoritmo *Q-Learning*. En este algoritmo, al ser libre de modelo, se parte de una colección de tuplas de experiencia. Como tenemos un estimador por acción necesitamos tantos conjuntos de tuplas como acciones, en donde cada conjunto refleja la experiencia recogida con cada acción. El algoritmo trata de refinar el estimador en cada iteración, llegando a converger a la función de valor y la política óptimas en el mejor caso. Del mismo modo que ocurría con *Smooth Value Iteration* dependiendo del tipo de aproximador utilizado (redes de neuronas, árboles de decisión, etc) el algoritmo podría converger a políticas no óptimas o no converger. El algoritmo se muestra en la figura 21.

- Dados
  - Un espacio de estados S y una zona de meta $G \subset S$
  - Un conjunto de acciones finito A,de tamaño L, donde $A = \{a_1, \ldots, a_L\}$
  - Una colección de tuplas de experiencia del tipo $\langle s, a_i, s' \rangle$, donde $s \in S$ es un estado desde el que se ejecuta la acción a_i, y s' es el estado al que se llega
  - Una función de coste $Coste: S \times A \rightarrow R$
  - L aproximadores de la función de valor-acción, $\hat{Q}_{a_i}: S \rightarrow R$
- $iter = 0$
- $q_{a_i}^0[j] = 0, \forall\, j = 1, \ldots, N, \quad a_i \in A$
- Repetir
  - Entrenar $\hat{Q}_{a_i}^{iter}$ para aproximar el conjunto de entrenamiento formado por: $\langle s_1, q_{a_i}^{iter}[1] \rangle, \ldots, \langle s_N, q_{a_i}^{iter}[N] \rangle$
  - $iter = iter + 1$
  - Para j=1 hasta N, y para todo $a_i \in A$:

$$q_{a_i}^{iter}[j] = \left\{ \begin{array}{cc} 0 & si\ s_j' \in G \\ mín_{a_r \in A} \{ Coste(s_j, a_j) + \hat{Q}_{a_r}^{iter-1}(s_j') \} & en\ otro\ caso \end{array} \right\}$$

Hasta que el vector q no cambie

- Devolver $\hat{Q}_{a_i}^{iter-1}$, para $i = 1, \ldots, L$

Figura 21: Algoritmo Iterative Smooth Q-Learning.

Fase 2: MDQL

En los problemas deterministas de programación dinámica se pueden usar varios criterios de particionado para discretizar de manera incremental el espacio de estados. Estos criterios buscan introducir más regiones en aquellas zonas donde la aproximación de la función de valor o la política es más difícil, por ejemplo, zonas donde existan discontinuidades. Estas discontinuidades pueden entenderse como zonas próximas ente si, donde se encuentran valores muy diferentes de la función que se intenta aproximar. El algoritmo ISQL intenta distinguir de la forma más clara posible las zonas de la función en donde se obtienen distintos valores, es decir, las regiones del espacio de estados con distintos valores de la función valor-acción.

La idea del algoritmo 2SRL consiste en que aunque ISQL obtenga políticas que puede no ser óptimas, si el aproximador utilizado nos proporciona, además de una aproximación, una discretización del espacio de estados, esta discretización podría definir de manera correcta los límites de las función de valor-acción.

La discretización obtenida en la primera fase (ISQL) puede ser usada en la segunda fase para refinar la función de valor obtenida por el aproximador o simplemente para aplicar la discretización en un algoritmo como Q-Learning inicializando la política a 0. Esta segunda fase en la que utilizamos la discretización de ISQL junto con Q-Learning se conoce como *Multiple Discretization Q-Learning* (MDQL) [Fernandez y Borrajo, 2008].

En la figura 22 podemos ver como trasladar el esquema de la primera fase al de la segunda. En este caso utilizamos como aproximador en la fase ISQL el algoritmo M5 rules [Quinlan, 1992] (1 aproximador por acción). La parte izquierda de las reglas servirá para definir las regiones en las que se dividirá el espacio de estados. A cada regla se le puede asignar un índice de la tabla Q que será utilizado para saber que valor de la tabla se debe actualizar en cada momento del algoritmo Q-Learning. La parte derecha de la regla definirá un valor que puede servir para inicializar la casilla correspondiente de la tabla Q. Si utilizáramos un árbol de decisión haríamos corresponder cada hoja del árbol con una región o casilla de la tabla Q. Numerando las hojas podemos indizar la tabla Q e inicializarla con el valor de clase con que la hoja estaba etiquetada originariamente (figura 23).

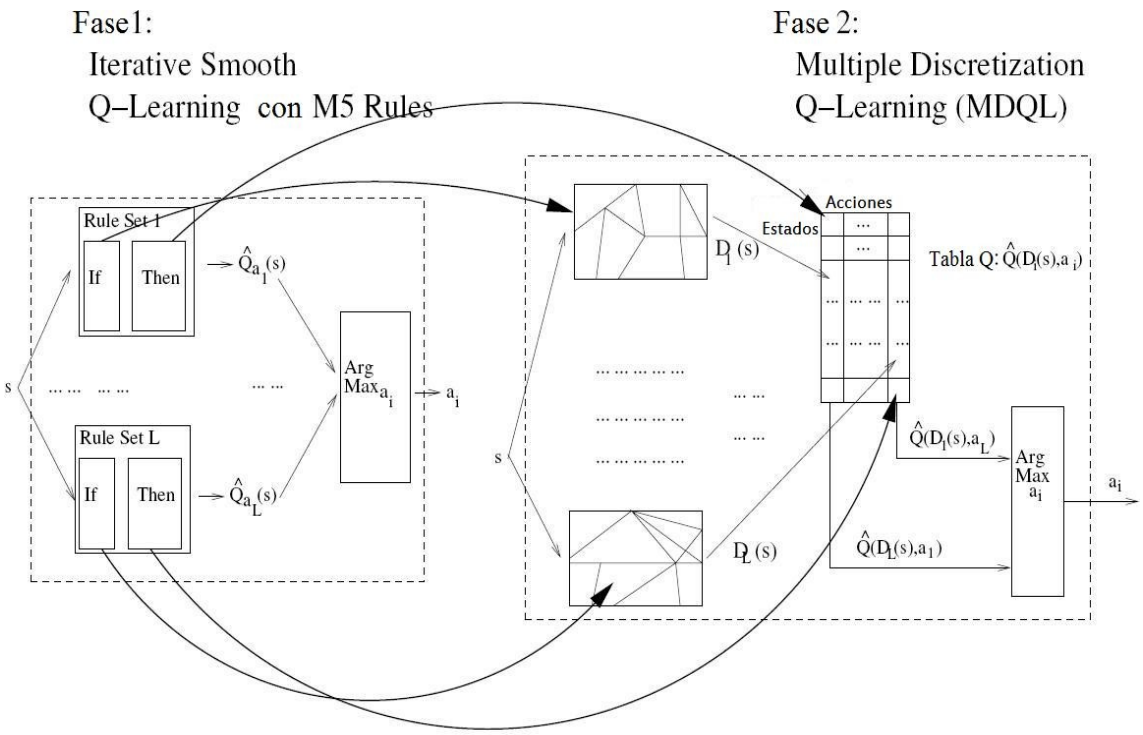

Figura 22: Paso del esquema ISQL al MDQL del algoritmo 2SRL utilizando como aproximador M5 rules.

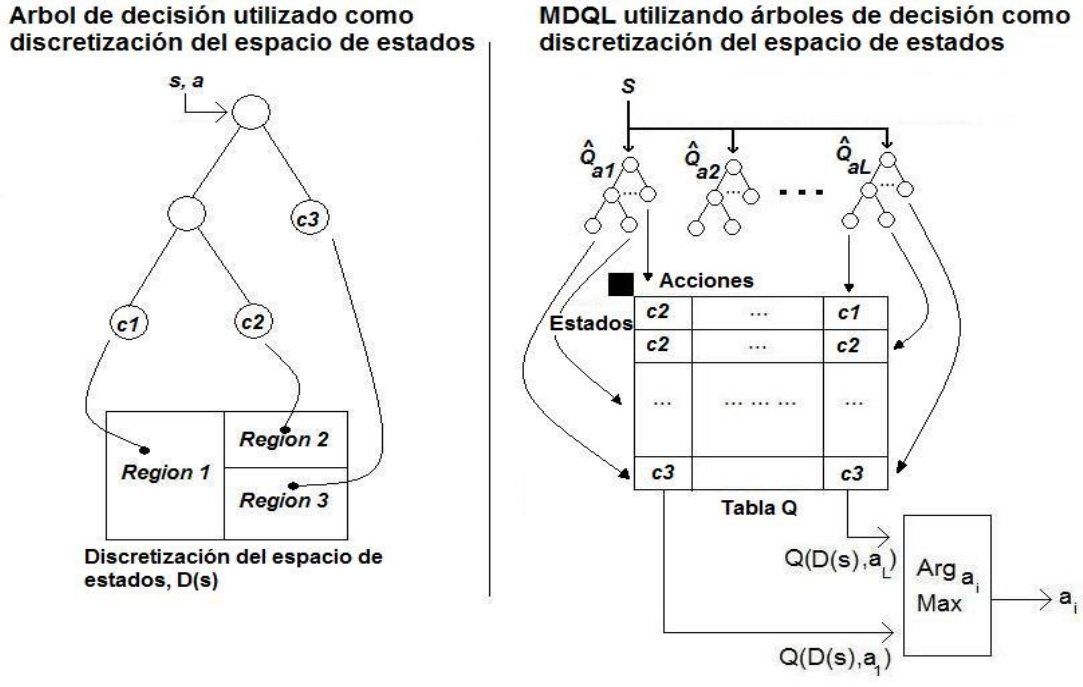

Figura 23: Paso del esquema ISQL al MDQL del algoritmo 2SRL utilizando como aproximador árboles de decisión.

2.3 Robocup Keepaway

La *Keepaway* es considerada como una subtarea de la *Robocup Soccer*. *RoboCup* es una iniciativa internacional de investigación y educación. Tiene la finalidad de promover la investigación en robótica y en el campo de la inteligencia artificial para lo cual se provee un problema estándar en donde un amplio abanico de tecnologías y técnicas pueden ser integradas y analizadas, las cuales pueden ser utilizadas posteriormente en proyectos reales. La *Keepaway* utiliza como base para su implementación el *Soccer Server* de la *Soccer simulation League,* que es una de las ligas más antiguas de la *RoboCup Soccer*. Esta liga consiste en dos equipos que juegan entre ellos sobre una red. El *Robocup SoccerServer* es un simulador de sotware que simula el juego y distribuye información sensorial a los agentes conectados. Cada participante conecta a dicho servidor 11 agentes y opcionalmente un agente entrenador. Cada agente funciona como un proceso independiente y el servidor hace de intermediario en las comunicaciones entre los agentes. Además, cada jugador debe indicar al servidor la acción que realizará afectando al entorno que le rodea.

El juego de la *Keepaway* consiste en un equipo que intenta mantener la posesión del balón en una región de juego limitada, mientras el equipo contrario intenta quitarles el balón. Los primeros son conocidos como *keepers* y los segundos como *takers*. Cuando los *takers* roban el balón a los *keepers* o el balón sale de la zona de juego se considera que el episodio ha finalizado y se inicia un nuevo episodio con los *keeper* en posesión del balón. Se pueden configurar el número de keepers y el de takers que formarán cada equipo y el tamaño de la región de juego.

Figura 24: Captura del monitor de *Keepaway* corriendo en el *soccerserver*.

Los agentes de la *RoboCup* reciben percepciones visuales cada 150ms indicando la distancia relativa y ángulos con el resto de objetos visibles en el terreno de juego, como la pelota y el resto de jugadores que participan. Los agentes pueden ejecuta acciones cada 100ms. A las distancias y ángulos percibidos se les añade un cierto ruido que hace que el agente no tenga medidas exactas.

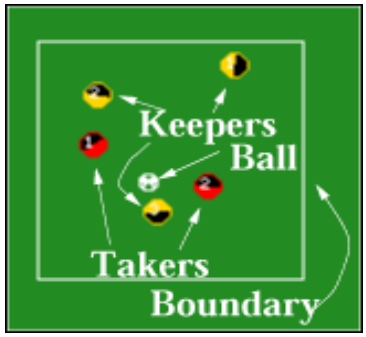

Figura 25: Tres *keepers* vs dos *takers* en *keepaway*.

Los agentes son controlados por procesos independientes y la comunicación entre ellos se realiza a través del *SoccerServer*. Desde el punto de vista del aprendizaje este hecho es de gran relevancia ya que los agentes no comparten su experiencia entre si, y tienen que aprender de manera individual.

En *keepaway*, un omnisciente *coach agent* controla el juego, finalizando los episodios cuando los *takers* logran obtener la posesión de la pelota o cuando la pelota sale fuera de la región de juego. En el comienzo de cada episodio el *coach* reparte a los jugadores y a la pelota por el terreno de juego semi-aleatoriamente. Todos los *takers* comienzan en la esquita inferior izquierda. Tres *keepers* elegidos aleatoriamente son situados en cada una de las esquinas restantes y el resto de *keepers,* si los hubiera, son colocados en el centro de la región de juego. La pelota se coloca al lado del *keeper* de la esquina superior izquierda. Cuando un episodio termina otro comienza dando lugar a una sucesión de episodios en donde cada jugador aprende independientemente del resto de sus compañeros. Para un jugador, cada episodio empieza cuando recibe la pelota por primera vez y termina cuando el equipo pierde la posesión de la pelota o ésta sale del área de juego.

Las acciones que los jugadores de la *keepaway* pueden ejecutar son la siguientes:

- *HoldBall()*: El jugador que ejecuta esta acción mantiene la posesión de la pelota procurando tenerla lo más alejada posible de los oponentes.
- *PassBall(k)*: Pasa la pelota al *keeper k*. Los *keepers* van numerados de forma relativa a la distancia con respecto al *keeper* que posee el balón. Así en 3vs2 si *k=1* el agente estará pasando el balón al *keeper* más cercano a él mismo. Si, en cambio, *k=2*, estará pasando el balón al compañero más alejado.
- *GetOpen()*: El jugador se mueve a una posición que está libre de oponentes.

- *GoToBall()*: El jugador intercepta un pase o se mueve hacia donde está el balón.
- *BlockPass(k)*: El jugador se mueve a la posición entre el *keeper* que tiene la posesión de la pelota y el *keeper k*.

Todas estas habilidades excepto *PassBall(k)* pueden ser resueltas en un ciclo del simulador. En cambio, *PassBall(k),* influye en el comportamiento del jugador durante varios pasos del simulador. Otro suceso que puede ocurrir es que el simulador pierda comandos y en ese caso el agente no pueda volver a ejecutar otra acción hasta pasados dos o más ciclos.

Keepers

Dependiendo de si el *keeper* tiene o no la pelota, éste puede ejecutar diferentes acciones. Cuando el keeper no está en posesión de la pelota la rutina que ejecuta se llama *receive* y funciona de la siguiente manera: Si un compañero de equipo posee la pelota, o puede conseguirla más rápido que el *keeper* que invoca esta función, entonces invoca a la primitiva *GetOpen*() durante un paso del simulador. En otro caso, invoca a la función *GoToBall*(). La invocación a esta rutina se repite hasta que el jugador se encuentra en posesión de la pelota o el episodio finaliza.

Cuando un *keeper* está en posesión del balón, puede elegir entre varias acciones a ejecutar. Por un lado puede retener el balón y por otro lado puede decidir pasarlo a un compañero de equipo. El número de acciones disponibles estará sujeto al número de compañeros que integran el equipo. En el caso de 3 *keepers* contra 2 *takers*, el *keeper* poseedor del balón puede elegir entre no pasar el balón y retenerlo, pasarlo a su compañero más cercano o pasarlo a su compañero más lejano. En total dispondría de 3 acciones a elegir. En el caso de ser 4 *keepers*, el poseedor del balón dispondría de 4 acciones. De forma general un *keeper* dispondrá del siguiente conjunto de acciones:

$$\{HoldBall, PassBall(k_2), PassBall(K_3), \ldots, PassBall(K_n)\}$$

donde $HoldBall$ retiene el balón y $PassBall(k_i)$ realiza un pase al compañero i, *pasando a continuación a ejecutar Receive* hasta que de nuevo se haga con el balón.

Como se ha mencionado anteriormente los *keepers* se numeran según la proximidad que tienen al *keeper* que posee el balón, siendo el *keeper* 1 el que posee el balón, el *keeper* 2 el más cercano al 1, el *keeper* 3 el segundo más próximo al *keeper* 1, etc.

Cada jugador sólo tiene el control de una pequeña porción del comportamiento del equipo. Una vez que pasa la pelota, el resto de acciones que ejecutan sus compañeros son decididas independientemente por ellos.

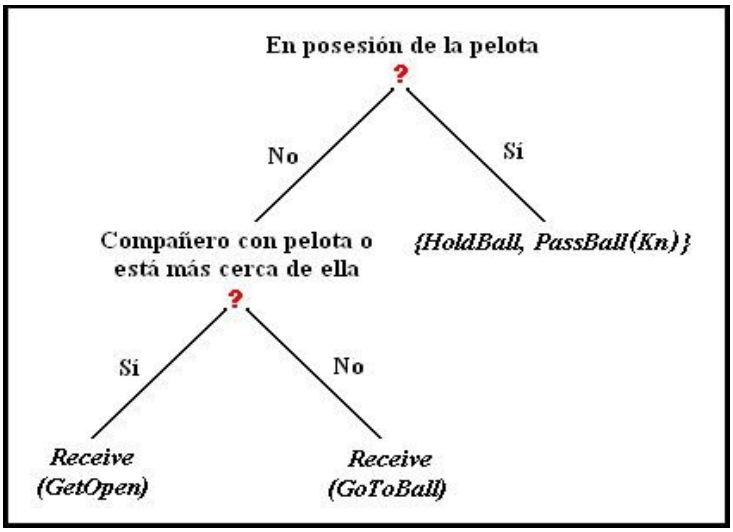

Figura 26: Esquema de decisión de acciones de un *keeper*.

La representación de los estados utilizada por los *keepers* les permite conocer información diversa del mundo que les rodea. Las variables que componen un estado son calculadas en función de las posiciones que ocupan los *keepers*, los *takers* (ordenados siempre en orden creciente respecto a la distancia al *keeper* poseedor del balón), y el centro de la región de juego. La tabla 1 resume las características de un estado para un juego 3vs2.

Característica	Descripción
dist(K_1,C)	Distancia del keeper poseedor del balón al centro de la región de juego.
dist(K_2,C)	Distancia del keeper 2 al centro de la región de juego.
dist(K_3,C)	Distancia del keeper 3 al centro de la región de juego.
dist(T_1,C)	Distancia del taker 1 al centro de la región de juego.
dist(T_2,C)	Distancia del taker 2 al centro de la región de juego.
dist(K_1,K_2)	Distancia entre el keeper 1 y el keeper 2.
dist(K_1,K_3)	Distancia entre el keeper 1 y el keeper 3.
dist(K_1,T_1)	Distancia entre el keeper 1 y el taker 1.
dist(K_1,T_2)	Distancia entre el keeper 1 y el taker 2.
Min(dist(K_2,T_1),dist(K_2,T_2))	Mínimo de la distancia entre el keeper más cercano al poseedor del balón y cada uno de los takers.
Min(dist(K_3,T_1),dist(K_3,T_2))	Mínimo de la distancia entre el keeper más alejado del poseedor del balón y cada uno de los takers.
Min(ang(K_2,K_1,T_1),ang(K_2,K_1,T_2))	Mínimo entre el ángulo (K_2,K_1,T_1) con vértice K_1 y el ángulo (K_2,K_1,T_2) con vértice K_1.
Min(ang(K_3,K_1,T_1),ang(K_3,K_1,T_2))	Mínimo entre el ángulo (K_3,K_1,T_1) con vértice K_1 y el ángulo (K_3,K_1,T_2) con vértice K_1.

Tabla 1: Características de un estado para un juego 3vs2.

A medida que aumentan el número de jugadores se incrementan el número de variables necesarias para describirlo. Así, en *keepaway* 3vs2 se emplean 13 variables, en *keepaway* 4vs3 se emplean 19 variables y en 5vs4 se emplean 25 variables. La figura 27 muestra gráficamente las variables de un estado de un *keeper* que tiene la posesión del balón.

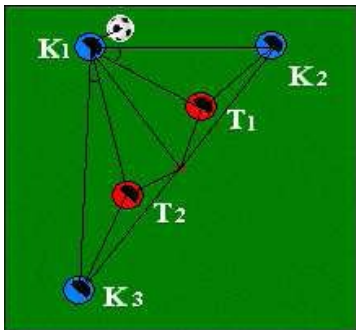

Figura 27: Variables de estado de un *keeper* en 3vs2.

Takers

El comportamiento de un *taker* cuando está en posesión del balón es mantener la posesión, invocando a la función *HoldBall*() durante un paso del simulador. Cuando no tiene el balón el *taker* puede elegir entre las siguientes acciones: *GoToBall*(),*BlockPass*(k_2),*BlockPass*(k_3),...,*BlockPass*(k_n). Cuando el balón no está en posesión de nadie, como sucede en un pase, el *keeper* etiquetado como *K1* será aquel a quien se cree que va destinado el pase. Los *takers* definen 3 políticas distintas cuando no tienen el balón:

- Ramdom-T: El *taker* elige aleatoriamente entre el conjunto de acciones que puede ejecutar.
- All-to-ball: Los *takers* siempre eligen la acción *GoToBall*().
- Hand-Coded-T: Si no hay otro *taker* que pueda alcanzar la pelota antes que el *taker* que evalúa esta condición, o si éste *taker* es el más cercano o el segundo más cercano a la pelota, entonces elige *GoToBall*(). Si no se cumple lo anterior ejecuta *BlockPass*(*k*), siendo *k* el *keeper* con mayor angulo con vértice en el balón y libre de marca.

El esquema de decisión según si el *taker* tiene o no el balón se muestra en la figura 28.

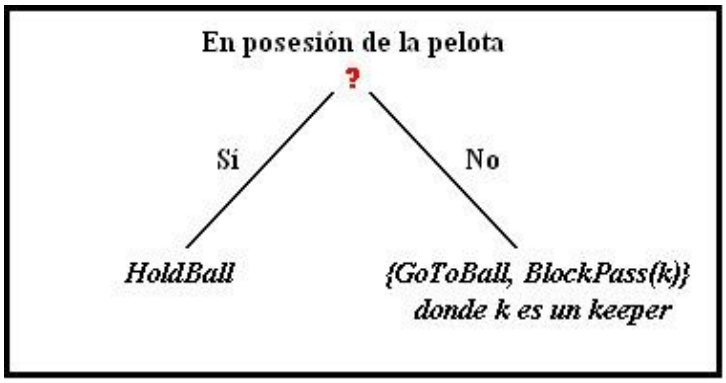

Figura 28: Esquema de decisión de acciones de un *Taker*.

2.4 Aprendizaje por refuerzo en Keepaway

Como se ha explicado en apartados anteriores, para poder aplicar técnicas de aprendizaje por refuerzo sobre un dominio, es necesario que exista un número finito de estados y acciones que permitan elaborar tablas de valores que guarden información sobre la relación estado-acción sin que esto implique un coste computacional elevado. En el dominio de la *Keepaway* nos enfrentamos al principal inconveniente de tener un espacio de estados infinito, ya que el simulador devuelve valores continuos de los atributos de los estados. En lo que respecta al estado de acciones, tenemos un conjunto reducido y limitado de acciones entre las que un agente puede elegir. Para un juego 3vs2 disponemos de solo tres acciones, *hold*, *pass*(*k1*) y *pass*(*k2*). El aprendizaje en *keepaway* se realiza de manera individual, es decir, cada agente dispone de su tabla Q y toma las decisiones en base a ella, sin tener en cuenta la información que hayan aprendido sus compañeros. El aprendizaje está orientado a los *keepers*, siendo el principal objetivo de este equipo mantener la posesión del balón el máximo tiempo posible. En el caso de los *takers*, éstos ejecutan un comportamiento fijo y preprogramado.

Descripción de los episodios

El aprendizaje en este dominio va a consistir en una sucesión de episodios que se ejecutan uno tras otro. La finalización de un episodio vendrá marcada por la recuperación del balón por parte de los *takers* o porque la pelota haya abandonado los límites del terreno de juego. Es posible que entre la ejecución de una acción y la siguiente tengan lugar varios pasos del simulador. Desde el punto de vista de un individuo un episodio consiste en una secuencia de estados, acciones y recompensas que tienen lugar durante la interacción del agente con el entorno. Esta secuencia tiene la forma:

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_i, a_i, r_{i+1}, \ldots, r_j, s_j$$

donde la acción a_i es seleccionada en cada momento en función de una presuntamente incompleta percepción del estado s_i , y donde s_j es un estado final en el cual los *takers* toman la posesión de la pelota o ésta sale del terreno de juego. La recompensa que reciban los *keepers* en cada momento vendrá determinada por cada paso de tiempo del simulador que logran mantener la posesión de la pelota. Por eso, la recompensa r_i se establece como el número de pasos del simulador que han transcurrido desde la ejecución de la última acción realizada por el *keeper*, a_{i-1}, hasta que el *keeper* vuelve a recibir la pelota y con ella la información del estado y el refuerzo r_i. De esta forma el refuerzo r_i se define como $r_i = t_i - t_{i-1}$. En definitiva el objetivo de cada *keeper* es seleccionar aquella acción que permita al equipo mantener la posesión de la pelota el mayor tiempo posible, maximizando de esta forma el refuerzo total.

El problema de la generalización de estados en Keepaway

Para poder aplicar aprendizaje por refuerzo en la *Keepaway* necesitamos poder expresar de manera discreta el conjunto de estados del dominio. El tener una representación discreta del espacio de estados nos permite representar la función de valor de forma tabular, algo necesario para aplicar las técnicas de aprendizaje vistas anteriormente. En el caso de discretizar los atributos de los estados simplemente mediante cuantificación escalar, en un juego 3vs2 tendríamos 13 atributos para representar un estado como se pudo ver en la sección 2.3. Si el agente sigue un comportamiento aleatorio se comprueba experimentalmente que para los 50196 estados que ha recorrido el *keeper*, cada atributo de los estados presenta las estadísticas que se muestran en la tabla 2.

Atributo	Rango	Media
dist(K_1,C)	[0.0518, 17.2131]	9.2583
dist(K_1,K_2)	[0.6, 23.0812]	13.3354
dist(K_1,K_3)	[5.0, 30.0]	18.7898
dist(K_1,T_1)	[0.0379, 24.9569]	9.9701
dist(K_1,T_2)	[0.1154, 27.1]	11.1564
dist(K_2,C)	[0.0984, 17.5724]	10.3482
dist(K_3,C)	[0.4496, 17.7348]	11.1644
dist(T_1,C)	[0.0117, 17.8569]	5.8220
dist(T_2,C)	[0.0384, 18.6991]	6.1655
Min(dist(K_2,T_1),dist(K_2,T_2))	[0.0, 34.0404]	13.0586
Min(dist(K_3,T_1),dist(K_3,T_2))	[0.1194, 24.7896]	13.8254
Min(ang(K_2,K_1,T_1),ang(K_2,K_1,T_2))	[0.0, 178.0]	60.4154
Min(ang(K_3,K_1,T_1),ang(K_3,K_1,T_2))	[0.0, 179.0]	40.8131

Tabla 2: Estadísticas de los valores de los componentes de un estado.

Realizando una discretización en intervalos de 3 metros para las medidas de distancia y de 10 grados para los ángulos, los niveles que requiere cada atributo pueden verse en la tabla 3.

Atributo	Niveles
dist(K_1,C)	6
dist(K_1,K_2)	7
dist(K_1,K_3)	8
dist(K_1,T_1)	8
dist(K_1,T_2)	9
dist(K_2,C)	6
dist(K_3,C)	6
dist(T_1,C)	6
dist(T_2,C)	6
Min(dist(K_2,T_1),dist(K_2,T_2))	11
Min(dist(K_3,T_1),dist(K_3,T_2))	8
Min(ang(K_2,K_1,T_1),ang(K_2,K_1,T_2))	18
Min(ang(K_3,K_1,T_1),ang(K_3,K_1,T_2))	18

Tabla 3: Niveles de discretización por atributo utilizando intervalos de 3 metros para las medidas de distancia y de 10 grados para los ángulos.

Si suponemos que el agente puede ejecutar 3 acciones, la tabla necesaria para almacenar los refuerzos requerirá de $6^5 * 7 * 8^3 * 9 * 11 * 18^2 * 3 \approx 2{,}7$ billones de posiciones, lo cual supone una aproximación inabordable para un modelo de aprendizaje por refuerzo.

Afortunadamente, existen técnicas de discretización más eficientes que podemos utilizar en el dominio de la *keepaway* como puede ser la cuantificación vectorial, CMAC, árboles de decisión, etc. En este proyecto se ha optado por utilizar el algoritmo *Two steps reinforcement learning* como técnica elegida para conseguir una discretización del espacio de estados eficiente y así poder aplicar técnicas de aprendizaje por refuerzo clásicas.

Para poder aplicar el 2SRL primero necesitamos disponer de un conjunto de tuplas que recogeremos registrando la secuencia

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_i, a_i, r_{i+1}, \ldots, r_j, s_j$$

de un agente *keeper* con política de selección de acción aleatoria durante la ejecución de la *keepaway*. El conjunto de tuplas del fichero cuando se juega 3vs2 se escribirá de la siguiente forma:

$$\begin{aligned}
&s_0[0], s_0[1], s_0[2], \ldots, s_0[12], a_0, s_1[0], s_1[1], s_1[2], \ldots, s_1[12], r_1 \\
&s_1[0], s_1[1], s_1[2], \ldots, s_1[12], a_1, s_2[0], s_2[1], s_2[2], \ldots, s_2[12], r_2 \\
&\qquad\qquad \ldots. \\
&s_i[0], s_i[1], s_i[2], \ldots, s_i[12], a_i, s_{i+1}[0], s_{i+1}[1], s_{i+1}[2], \ldots, s_{i+1}[12], r_{i+1} \\
&\qquad\qquad \ldots.
\end{aligned}$$

donde cada $s_i[k]$ es la abreviatura de una característica o atributo que define el estado s en el instante i. La correspondencia de abreviaturas y atributos se define en la tabla 4.

Atributo	Abreviatura
dist(K_1,C)	s[0]
dist(K_1,K_2)	s[1]
dist(K_1,K_3)	s[2]
dist(K_1,T_1)	s[3]
dist(K_1,T_2)	s[4]
dist(K_2,C)	s[5]
dist(K_3,C)	s[6]
dist(T_1,C)	s[7]
dist(T_2,C)	s[8]
Min(dist(K_2,T_1),dist(K_2,T_2))	s[9]
Min(dist(K_3,T_1),dist(K_3,T_2))	s[10]
Min(ang(K_2,K_1,T_1),ang(K_2,K_1,T_2))	s[11]
Min(ang(K_3,K_1,T_1),ang(K_3,K_1,T_2))	s[12]

Tabla 4: Abreviaturas de los atributos que componen un estado de *keepaway* 3vs2.

Estos ficheros de tuplas pueden ser utilizados como entrada del algoritmo ISQL, lo que nos permite aplicar 2SRL sobre la *keepaway*.

Políticas de partida y comportamientos preprogramados

En *keepaway* es posible el uso de cualquier número de *keepers* y de *takers*. En nuestros experimentos nos centraremos en el escenario 3vs2, en el que participan 3 *keepers* y 2 *takers*. En esta modalidad de juego los dos *takers* involucrados siempre ejecutan la acción *GoToBall* y nunca la acción *BlockPass(k)* lo que hace que se simplifique el problema.

A la hora de evaluar los resultados que se puedan obtener, es muy conveniente tener un punto de partida o una referencia que nos ayude a determinar cómo de buena es la política que se consigue con aprendizaje por refuerzo frente a políticas preprogramadas o estrategias sencillas. A continuación se dan a conocer algunas de las políticas de partida más destacables y los resultados que obtienen en el entorno 3vs2. Cabe destacar que el tamaño de campo de juego utilizado es 25x25 y no se configuran restricciones de visión en los *keepers*.

Estrategias preprogramadas:

- *Random*: política de acción de fácil implementación consistente en elegir una acción aleatoria de forma uniforme.
- *Hold*: el *keeper* que se apodera de la pelota y la retiene indefinidamente sin realizar nunca un pase a un compañero.
- *Hand-coded*: es un comportamiento más elaborado que los anteriores en el que el *keeper* que recibe la pelota la mantiene en su posesión siempre que los *takers* se mantengan alejados. En el caso de que un *taker* se acerque pasará la pelota al compañero *keeper* más abierto. La forma en que se evalúa cuál de los compañeros es el más abierto consiste en una combinación lineal entre la distancia de los compañeros a su defensor más próximo y el ángulo entre el *keeper* con la pelota y el oponente más cercano a la línea de pase. La importancia relativa de estas dos características es ponderada por un coeficiente α. Si el compañero de equipo más abierto obtiene un valor por encima de cierto parámetro, β, entonces el *keeper* con la pelota se la pasa a este jugador. En caso contrario, el *keeper* retiene la pelota durante un paso más del simulador.

Las políticas descritas anteriormente han sido puestas en práctica en el simulador obteniendo los resultados visibles en la figura 29. Esta gráfica muestra los resultados en media que cada política obtiene.

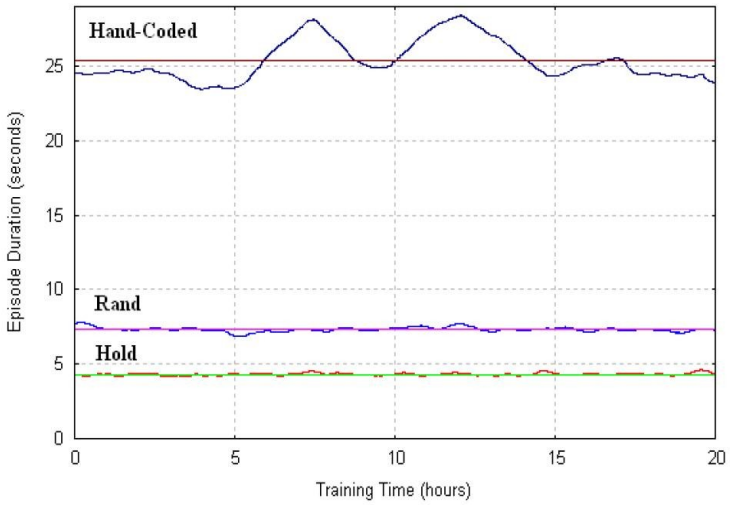

Figura 29: gráficas de duración media de los episodios en diferentes estrategias preprogramadas.

El objetivo del presente proyecto es conseguir mediante el algoritmo 2SRL políticas que mejoren considerablemente los resultados de las políticas más simples y se acerquen lo máximo posible o superen a las mejores políticas preprogramadas como *Hand-Coded*.

Trabajos anteriores de aprendizaje por refuerzo en Keepaway

Existen otros trabajos que han aplicado técnicas de aprendizaje por refuerzo en el dominio de la *keepaway*.

En [Stone et al, 2005] se aplicó con éxito funciones de aproximación lineal *tile-coding* (también conocida como CMAC) y Sarsa(λ). En este artículo planteaban unas *tilings* infinitas en extensión. El número de posibles *tiles* es muy elevado, sin embargo sólo un número relativamente pequeño de ellas es visitado. Los autores utilizan *hash-coding* para determinar qué *tiles* se encuentran activas en cada momento.

En [Fernández y Veloso, 2005] se presenta la reutilización de políticas como la búsqueda del equilibrio entre la explotación de la política actual que se está aprendiendo, la exploración del entorno mediante la ejecución de acciones aleatorias y la explotación de políticas pasadas. El anterior artículo no utilizó la *keepaway* como banco de pruebas pero en [Fernández, 2006] se emplean nuevamente las técnicas de reutilización de políticas en el dominio *keepaway*. Como técnica de discretización del espacio de estados se emplea cuantificación vectorial y como algoritmo de aprendizaje por refuerzo *Q-learning* (VQQL).

En [Taylor y Stone, 2005] se presenta la técnica de transferencia de comportamientos para acelerar el aprendizaje de los métodos de diferencia temporal en aprendizaje por refuerzo, transfiriendo la función de valor aprendida de una tarea a otra. Los experimentos los realizan utilizando el dominio *keepaway*.

En [Soni y Singh, 2006] los autores también estudian el problema de transferencia de comportamientos en el aprendizaje por refuerzo utilizando como escenario de los experimentos el mundo de los bloques y *keepaway*.

En [García et al, 2007] se revisaron las técnicas VQQL [Fernandez y Borrajo, 2000] y CMAC [Stone et al, 2005] aplicadas al dominio de la *keepaway* combinadas con la estrategia de exploración *ε-greedy* y reutilización de políticas utilizando la estrategia *Π-reuse*.

3 Contenido

En este apartado se expone el planteamiento teórico y el diseño que permite la aplicación del algoritmo 2SRL sobre el entorno de la *keepaway*. En primer lugar se exponen de forma teórica las diversas posibilidades que existen para implementar 2SRL en dominios con recompensa continua. En segundo lugar se explica el diseño del algoritmo ISQL para la *keepaway*, el cual utiliza como datos de entrenamiento un fichero log de tuplas generado durante la ejecución del juego. En tercer lugar se explica el diseño de la arquitectura de evaluación, que permite probar en la *keepaway* los aproximadores y discretizaciones del espacio de estados obtenidos como salida de la ejecución del ISQL.

3.1 Planteamiento teórico

El planteamiento inicial de 2SRL está pensado para dominios con recompensas discretas. El disponer de recompensas discretas permite poder usar algoritmos de clasificación para generar aproximadores de la función Q. Sin embargo, existen muchos dominios en donde la recompensa se expresa de forma continua, como en el caso de la *robocup keepaway*. Como solución se puede optar entre dos caminos:

- Por un lado podemos discretizar el espacio de recompensas y utilizar el algoritmo 2SRL original con algoritmos de clasificación. El problema de esta solución es que tendremos que discretizar manualmente y probar las diferentes discretizaciones hasta dar con una discretización del espacio de recompensas que funcione bien.

- Por otro lado podemos adaptar ISQL para utilizar algoritmos de regresión que generen una discretización del espacio de estados, la cual será usada en la fase MDQL. En este sentido podemos utilizar M5 [Quinlan, 1992] o PART [Frank y Witten, 1998] como aproximador de funciones.

Otra posible variación de 2SRL para adaptarlo a dominios con recompensa continua consiste en utilizar en la fase ISQL la función de actualización de Q-Learning para dominios estocásticos, la cuál se muestra a continuación:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \Upsilon \max_{a'} Q(s',a') - Q(s,a)] .$$

Esta función se puede utilizar en lugar de la función para dominios deterministas, que no es más que la anterior función donde la constante α toma el valor de 1. La función determinista se puede ver a continuación:

$$Q(s,a) \leftarrow r + \Upsilon \max_{a'} Q(s',a') .$$

El algoritmo ISQL adaptado para recompensas continuas y utilizando la función de actualización para dominios estocásticos se puede ver en la figura 30. Esta versión busca maximizar la recompensa recibida.

- Dados
  - Un espacio de estados S
  - Un conjunto de acciones finito A, de tamaño L, $A = \{a_1, \ldots, a_L\}$
  - Una colección de tuplas de experiencia del tipo $\langle s, a_i, s', r \rangle$, donde $s \in S$ es un estado desde el que se ejecuta la acción a_i, s' es el estado al que se llega, y r es la recompensa recibida
  - L aproximadores de la función de valor-acción, $\hat{Q}_{a_i} : S \rightarrow R$
- $iter = 1$
- $q_{a_i}^0[j] = 0, \forall\ j = 1,\ldots,N,\ \ a_i \in A$
- Repetir
  - Entrenar $\hat{Q}_{a_i}^{iter}$ para aproximar el conjunto de entrenamiento formado por: $\langle s_1, q_{a_i}^{iter}[1] \rangle, \ldots, \langle s_N, q_{a_i}^{iter}[N] \rangle$
  - $iter = iter + 1$
  - Para j=1 hasta N, y para todo $a_i \in A$:
    - Se actualiza $q_{a_i}^{iter}[j]$ usando la tupla $\langle s_j, a_j, s_j', r_j \rangle$:

$$q_{a_i}^{iter}[j] = (1-\alpha) q_{a_i}^{iter-1}[j] + \alpha [r_j + \Upsilon \max_{a'} \hat{Q}_{a'}^{iter-1}(s_j')]$$

Hasta que el vector q no cambie

- Devolver $\hat{Q}_{a_i}^{iter-1}$, para $i = 1,\ldots,L$

Figura 30: *Iterative Smooth* Q-*Learning* para dominios con recompensa continua.

Al igual que sucede con las técnicas de clasificación, las técnicas de regresión utilizadas en la fase ISQL deben aprender el aproximador de Q por medio de la división del espacio de estados en regiones. De esta manera podremos utilizar esta división como discretización del espacio de estados en la fase MDQL. En la figura 31 podemos ver como interpretar un árbol de regresión como discretización del espacio de estados y utilizar esta discretización para obtener una representación tabular de la función Q que nos permita aplicar técnicas de aprendizaje por refuerzo en la segunda fase de 2SRL.

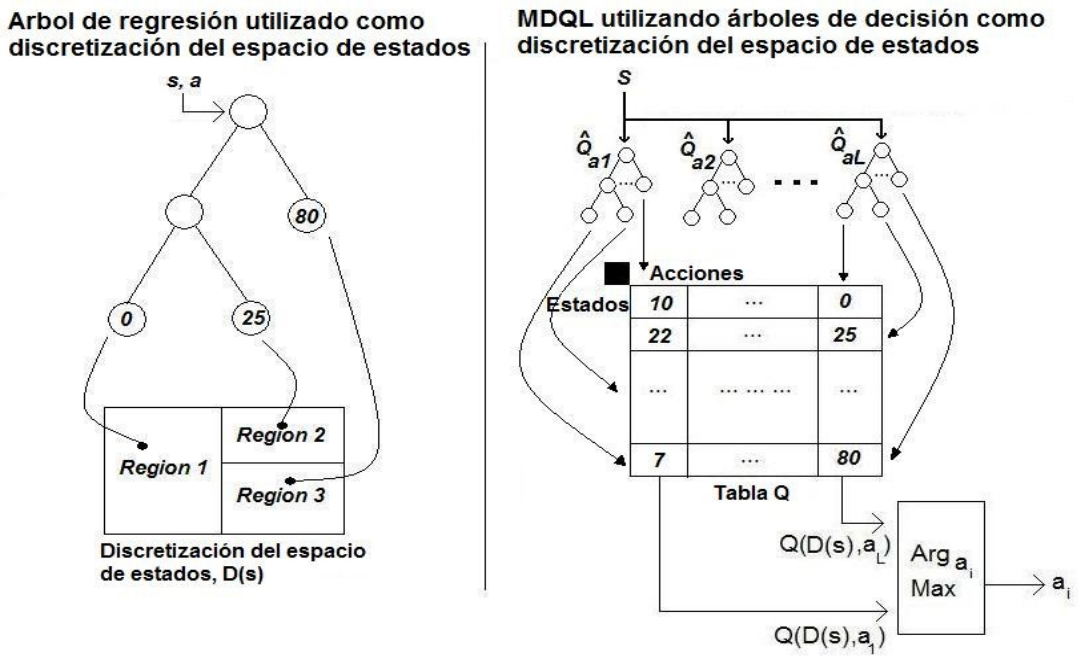

Figura 31: Paso del esquema ISQL al MDQL del algoritmo 2SRL utilizando como aproximador árboles de regresión.

De igual forma que en los árboles de decisión o clasificación, cada hoja del árbol se corresponde con una región del espacio de estados del dominio. La principal diferencia reside en que los árboles de regresión contienen un valor numérico en sus hojas, y no un valor de clase.

Los aproximadores de la función Q que se utilizan en 2SRL se definen de manera que reciban como entrada los atributos del estado s_t (ver tabla 4) y devuelva como salida r_{t+1}, siendo éste el máximo valor esperado de la recompensa cuando el agente se encuentra en el estado s_t. Como se explicó en el apartado 2.2.3 en la sección referente a la aproximación de funciones, a la hora de generan un aproximador de la función de valor-acción, Q(s,a), podemos optar por dos alternativas:

- Incluir la acción como parámetro de entrada del aproximador junto con los atributos del estado. De este modo se utiliza un aproximador único para todas las acciones.

- Generar un aproximador por cada posible acción a ejecutar, generando un total de *N* aproximadores, donde *N* es el número de acciones. En este caso utilizamos un aproximador múltiple.

La figura 32 muestra las distintas posibilidades para obtener un aproximador de la función Q en la *keepaway*.

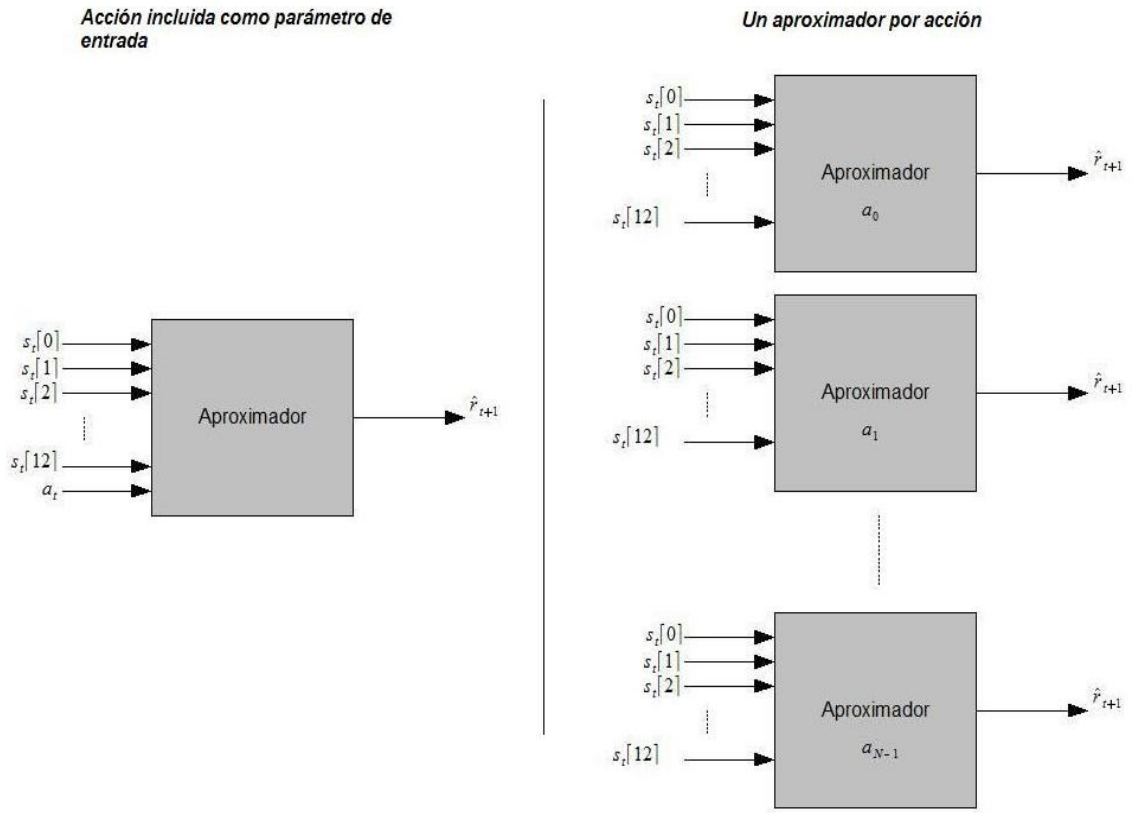

Figura 32: Aproximadores del refuerzo recibido según el estado y la acción utilizando un aproximador único y múltiple.

La tabla 5 resume las diferentes variables de configuración del algoritmo 2SRL que se pueden inferir a partir del planteamiento teórico expuesto.

Configuración de 2SRL		
Parámetro	Valores	
Aproximador	Único	Múltiple
Algoritmo de aproximación	Clasificación	Regresión
Función de actualización de Q en ISQL	Determinista $\alpha = 1$	Estocástico $0 < \alpha < 1$

Tabla 5: Parámetros de configuración del algoritmo 2SRL.

El segundo paso de 2SRL consiste en utilizar los estimadores generados por ISQL, no como aproximadores de la función Q, sino como discretización del espacio de estados del dominio del problema. 2SRL sugiere que la división de regiones de los aproximadores generados, en este caso en forma de árboles, puede funcionar bien como forma de simplificar espacios de estados continuos o muy extensos. La idea es que el criterio de división del espacio de estados

vendrá dado por el valor de la función Q. De esta manera aplicamos un método de discretización supervisado al contrario que ocurre con otros métodos de discretización del espacio de estados como, por ejemplo, VQ.

La forma de utilizar las regiones definidas por los árboles de clasificación /regresión para conseguir una discretización es la siguiente:

1. Disponemos del árbol de clasificación/regresión original, que dado un estado y una acción nos da la previsión de la función de valor para ese estado y acción.
2. Numeramos cada uno de los nodos hoja del árbol de manera secuencial, haciendo que cada nodo hoja disponga de un número que lo identificará de manera unívoca del resto de hojas del árbol.
3. Por otra parte, generamos la tabla Q que tendrá tantas celdas o posiciones como hojas tiene el árbol de clasificación/regresión.
4. A cada celda de la tabla se le hace corresponder de forma unívoca una hoja del árbol. La idea es hacer que el identificador de cada nodo hoja funcione como índice de la tabla Q.
5. Inicializamos cada celda de la tabla Q con el valor de la función Q con el que estaba etiquetado su correspondiente nodo hoja del árbol.

Con este método conseguimos dividir el espacio de estados en tantas regiones como nodos hojas tiene el árbol aproximador. El árbol se convierte así en un direccionador de la tabla Q que nos indica en cada momento que celda de la tabla tenemos que consultar o actualizar dados un estado y una acción. Además, si inicializamos la tabla Q de la forma explicada anteriormente, es decir, con los valores de las hojas del árbol, no perdemos la política aprendida por ISQL. La figura 33 muestra de forma gráfica como se direcciona la tabla Q mediante un solo estimador donde la acción es un parámetro de entrada más del estado y con 3 estimadores suponiendo que el número de acciones posible para el agente son 3 y se ha generado un estimador por acción.

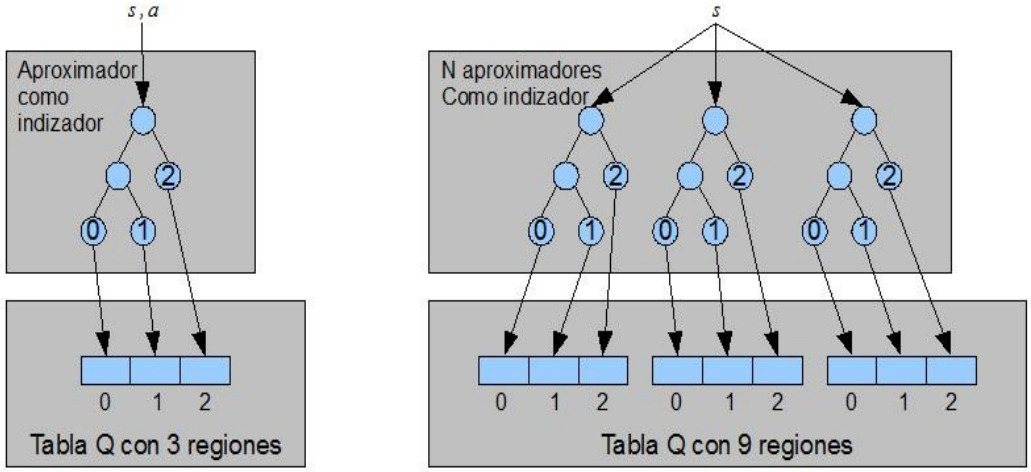

Figura 33: Utilización de aproximadores como direccionadores de la tabla Q.

Una vez expuesto todo el planteamiento teórico es necesario hacer un pequeño análisis de las características que deben cubrir los diseños del algoritmo ISQL y los modelos de agente necesarios para evaluar los aproximadores obtenidos y las discretizaciones del espacio de estados generadas a partir de éstos.

En el caso de ISQL necesitamos diseñar un sistema que acepte el uso de un aproximador único y un aproximador múltiple. Para ésto puede ser necesario definir un diseño diferente de ISQL para cada uno de los dos casos. Además, El diseño de ISQL debe tener la capacidad de poder aceptar tanto árboles de clasificación como árboles de regresión para la aproximación de la función Q. Por último, también será muy deseable poder utilizar la función de actualización para dominios deterministas y para dominios estocásticos. Este último punto es fácil de cubrir simplemente haciendo configurable el valor de la variable α.

Para evaluar los aproximadores de la función Q obtenidos en la fase ISQL y las discretizaciones del espacio de estados generadas a partir de ellos en la fase MDQL es necesario diseñar varios modelos de agente según la fase de 2SRL que queremos evaluar y el tipo de aproximador utilizado. La tabla 6 muestra los diferentes modelos necesarios para obtener una arquitectura que permita una evaluación completa de 2SRL.

Fase \ Aproximador	Único	Múltiple
ISQL	ISQL-UNICO	ISQL-MULTIPLE
MDQL	MDQL-UNICO	MDQL-MULTIPLE

Tabla 6: Tabla resumen de los modelos de agente necesarios para la evaluación de las distintas fases de 2SRL.

3.2 Diseño del ISQL

A continuación se expone el diseño modular orientado a facilitar la implementación del algoritmo ISQL. La idea consiste en proporcionar a un sistema un fichero de tuplas de experiencia como entrada y que éste modifique las tuplas de experiencia actualizando el refuerzo asociado en cada iteración del algoritmo. En la actualización de los refuerzos se utiliza un aproximador de la función de refuerzo que nos permite determinar cual es el refuerzo futuro esperado en el siguiente estado transitado. Este aproximador es calculado con las tuplas de experiencia modificadas en la iteración anterior. Las tuplas de experiencia que se esperan a la entrada son del tipo $\langle s, a, s', r \rangle$,donde s es el

estado en el que se encuentra el agente, a es la acción ejecutada en el estado s, s' es el estado al que se ha transitado desde s ejecutando la acción a, y r es el refuerzo recibido tras haber ejecutado la acción a. La salida última del algoritmo es el aproximador utilizado en la última iteración, que aproximará de forma más precisa la función de valor óptima.

Después de un análisis de las diferentes fases del algoritmo podemos diferenciar distintos módulos que pueden ser implementados de forma independiente:

- Módulo actualizador de tuplas: en cada iteración, un módulo software debe encargarse de leer el fichero de tuplas y aplicar la ecuación de actualización de Q-Learning a los refuerzos. Como resultado el módulo debe generar un fichero de salida con las mismas tuplas que el fichero de entrada, pero con los refuerzos actualizados. Este módulo debe hacer uso del aproximador generado en la iteración anterior para realizar las actualizaciones. Cuando se ejecuta el módulo en la primera iteración no se dispone de un aproximador. Por lo tanto, la primera actualización se realiza con un aproximador inicial que devuelve cero en cualquier caso. Si el fichero de entrada contiene tuplas del tipo $\langle s, a_i, s', r_t \rangle$, el fichero de salida contendrá las mismas tuplas con el nuevo refuerzo $\langle s, a_i, s', r_{t+1} \rangle$, donde r_{t+1} para la ecuación estocástica de Q-Learning es:

$$r_{t+1} = r_t + \alpha[r_0 + \gamma \max_{a'} \hat{R}_t(s', a') - r_t]$$

  Y para la ecuación determinista de Q-Learning, es decir, la ecuación estocástica con $\alpha = 1$ es:

$$r_{t+1} = r_0 + \gamma \max_{a'} \hat{R}_t(s', a')$$

  Si observamos las anteriores ecuaciones, para ambos casos se necesita siempre el fichero de tuplas original del que recuperar r_0, es decir, el refuerzo real que el entorno proporcionó al agente. Para el caso de la ecuación estocástica, además necesitamos conservar un fichero que guarde la actualización de la iteración anterior r_t. Por último, $\hat{R}_t(s, a)$ representa el aproximador de r_t, calculado con las tuplas $\langle s, a_i, s', r_t \rangle$.

- Módulo generador del aproximador: una vez que tenemos generado el fichero de tuplas actualizadas, necesitamos generar con la ayuda de éstas tuplas un aproximador que realice estimaciones de las recompensas. Para la generación del modelo de clasificación o regresión se necesita de un software de minería de datos que dado el conjunto de tuplas de entrada, nos genere fichero de salida con el modelo resultante. El paquete *Weka* cubre las necesidades que el algoritmo ISQL necesita para este proyecto. En concreto, se hará uso del algoritmo *J48* de clasificación y del *M5* de regresión con sus dos variantes, modelos *M5P* y regresión *M5P -R*.

- Módulo parser del modelo: una vez obtenido el modelo en el formato de salida del módulo generador, necesitamos transformar ese modelo en un código de alto nivel que pueda ser compilado junto con el código del módulo actualizador, haciendo posible el uso del modelo dentro del módulo actualizador de la siguiente iteración.

- Módulo compilador: una vez que tenemos el modelo traducido a un fichero de alto nivel compilable, se necesita hacer una recompilación del módulo actualizador junto con el modelo parseado. El resultado será el módulo actualizador compilado con el nuevo aproximador. Este módulo compilado será usado en la siguiente iteración del algoritmo ISQL.

La figura 34 plasma la interacción entre los diferentes módulos, las entradas y las salidas de cada módulo.

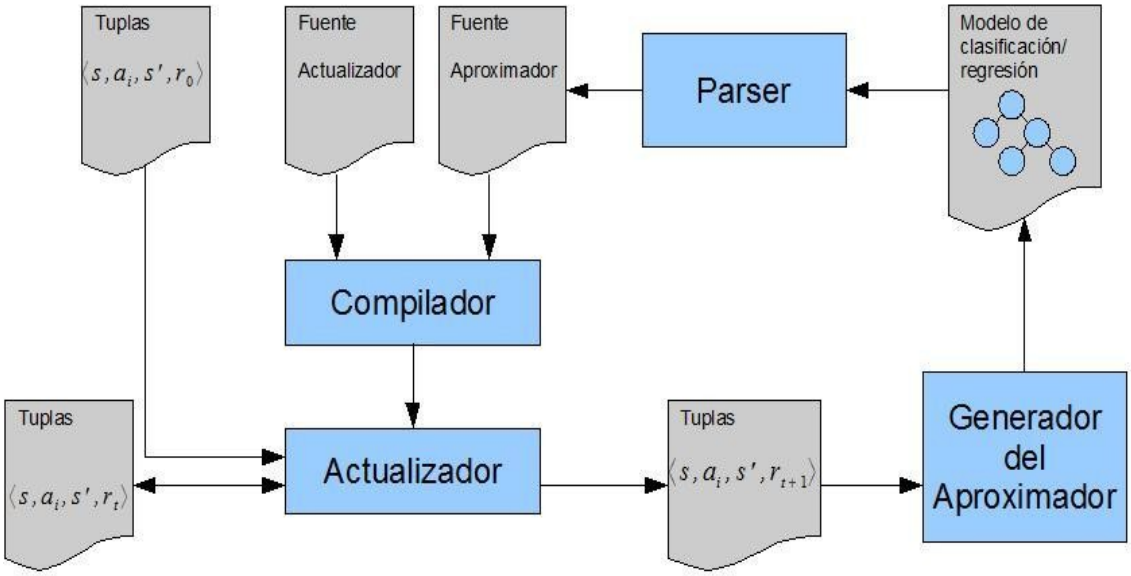

Figura 34: Diseño modular de ISQL

Los experimentos a realizar requieren de varios datos generados en la ejecución del ISQL para su correcta evaluación. Por ejemplo, nos interesa saber la distancia entre los refuerzos de las tuplas de una iteración y los refuerzos de las tuplas de la iteración anterior. Además es muy interesante poder recoger el número de regiones que cada aproximador genera en cada iteración. Estos ficheros *log* pueden ser incluidos como salidas del algoritmo. Además no solo nos interesa el último aproximador generado. También nos interesa medir la evolución de la bondad del aproximador en cada iteración. Por eso es interesante guardar los ficheros de tuplas de cada iteración para poder así regenerar el modelo y el aproximador. El guardar estos ficheros intermedios de la ejecución permite generar variaciones de los aproximadores con parámetros cambiados en el cálculo del modelo, como puede ser el factor de

poda, el número de instancias por hoja mínimo, etc. La figura 35 muestra el esquema de ISQL con los ficheros de entrada y salida generales.

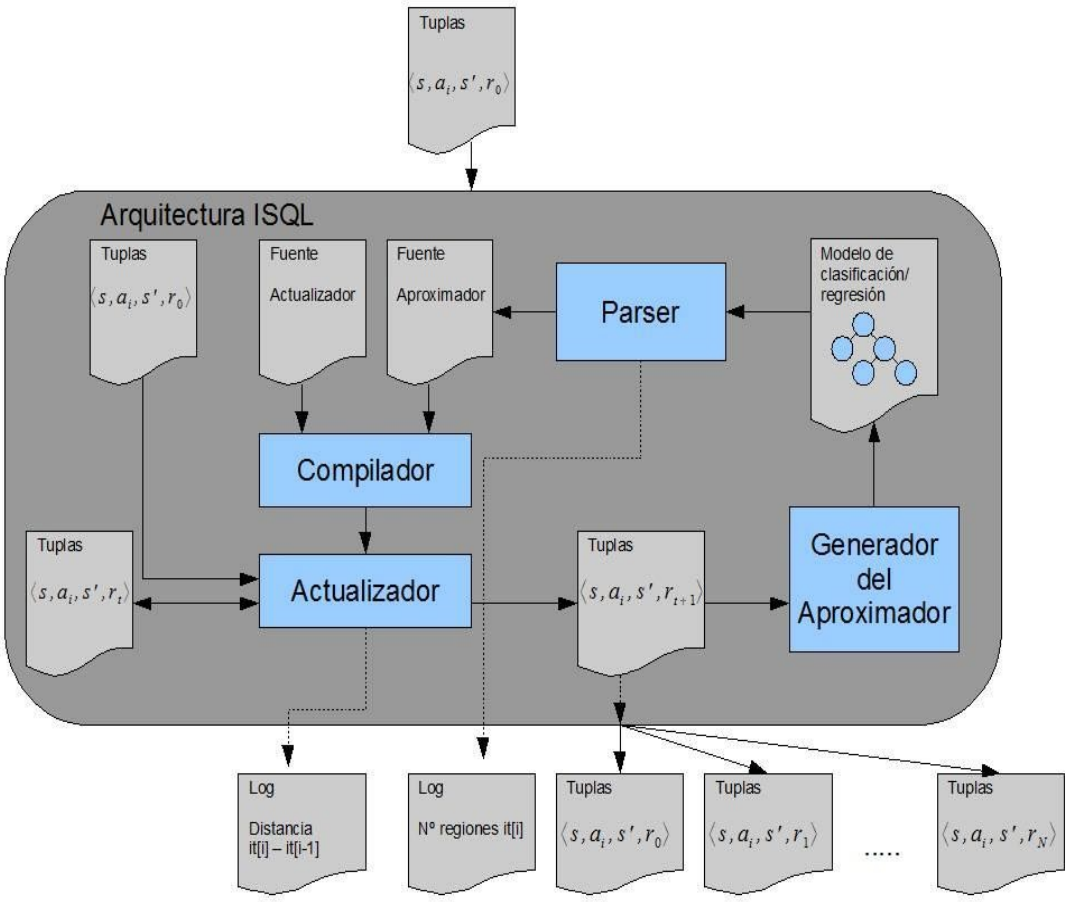

Figura 35: Especificación de ficheros E/S del diseño modular de ISQL

Con el conjunto de ficheros de salida de tuplas de la figura 35 podemos recalcular el aproximador que nos interese evaluar en el simulador de la *keepaway*. Si queremos evaluar, por ejemplo el aproximador que generó el algoritmo en la iteración t, solo tenemos que utilizar los módulos *Generador* y *Parser* para obtener un código fuente que podamos importar y utilizar en el controlador del agente de la *keepaway* a partir del conjunto de tuplas $\langle s, a_i, s', r_t \rangle$. Si diseñamos el módulo *Parser* para que genere, además del aproximador como lo define el modelo de clasificación/regresión, una variación del aproximador que devuelva el índice correspondiente a la región en la que se clasifica cada instancia de entrada, tal y como se explica al final de la sección 3.1, podemos evaluar el aproximador como discretizador en la fase MDQL. El esquema que debemos seguir para conseguir, a partir de las tuplas generadas en cada iteración de ISQL, un aproximador que podamos evaluar en la arquitectura de evaluación (sección 3.3) se muestra en la figura 36.

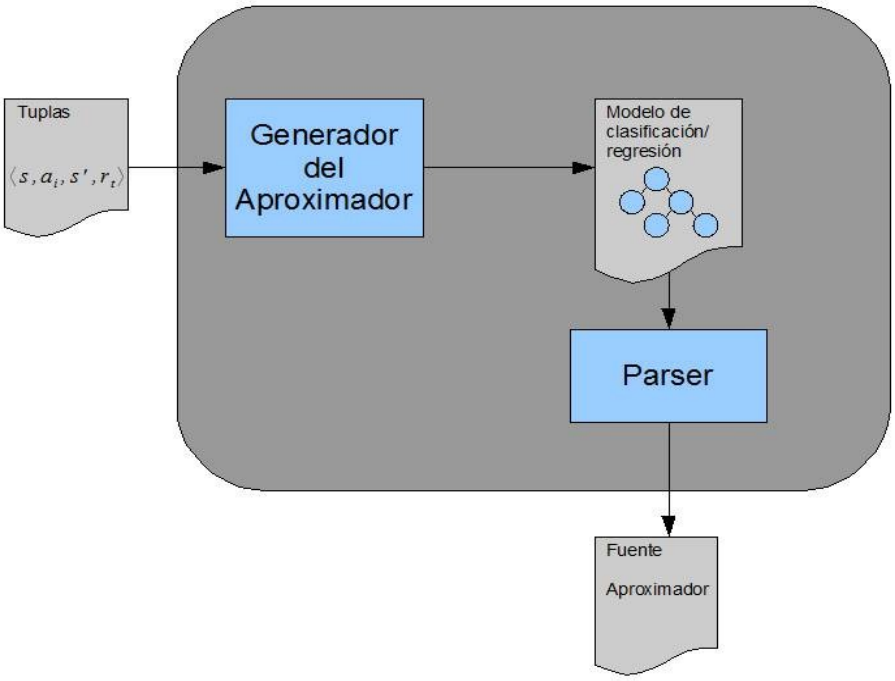

Figura 36: Recuperación del aproximador a partir de las tuplas de una iteración de ISQL.

Los anteriores esquemas corresponden de forma más directa a la implementación de ISQL con aproximador único. Para aplicar el algoritmo ISQL con un aproximador para cada acción, llamado en la sección 3.1 aproximador múltiple, el esquema es muy similar al visto anteriormente. La principal diferencia es que en vez de un solo fichero de tuplas, necesitamos tener un fichero de tuplas por cada acción. Así si tenemos 3 acciones necesitamos 3 ficheros, donde cada fichero contiene las tuplas de la acción correspondiente. El módulo *actualizador* tendrá que procesar cada fichero de tuplas de forma secuencial, generando por cada fichero de entrada, su correspondiente fichero de salida con los refuerzos actualizados. Además, también generará un fichero *Log* por cada fichero de acción de entrada. De la misma forma el *Generador* tendrá que recibir, uno por uno, los ficheros de tuplas actualizados con los que generará un modelo por cada acción que tendrá que ser "parseado". El *parser* generará un aproximador por cada modelo recibido a la entrada que serán compilados junto con el código base del *actualizador*. El *parser* también tendrá un *Log* del número de regiones por cada acción procesada. El esquema modificado para 3 acciones se muestra en la figura 37.

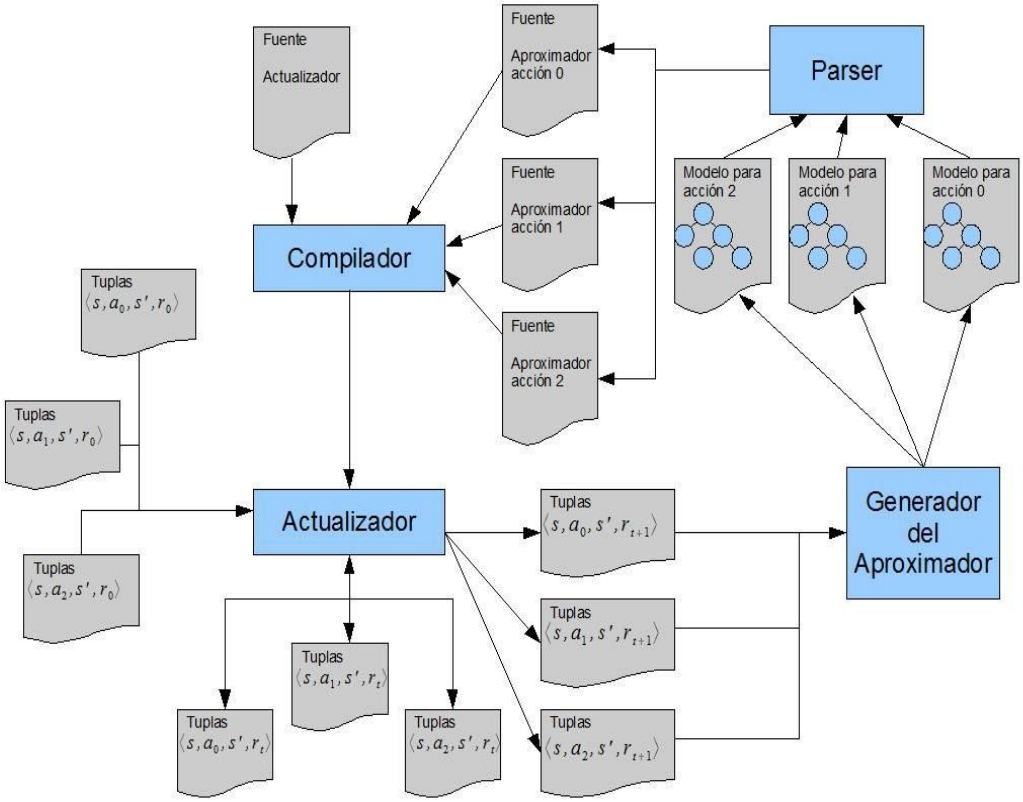

Figura 37: Diseño modular de ISQL utilizando aproximador múltiple con 3 aproximadores.

3.3 Diseño de la arquitectura de evaluación

A continuación se exponen los modelos que serán la base para la construcción del entorno de pruebas que medirá la bondad de los aproximadores obtenidos por el algoritmo ISQL. En primer lugar se profundiza en el diseño de la arquitectura que permitirá a un agente de la *keepaway* utilizar un árbol de clasificación/regresión como aproximador de la función Q. En segundo lugar se explica el diseño de los modelos que permiten utilizar las regiones definidas por un árbol de clasificación/regresión como discretización del espacio de estados en la fase MDQL. Los modelos expuestos se corresponden con los modelos reflejados en la tabla 6 del apartado 3.1.

3.2.1 Arquitectura de evaluación de ISQL

Para poder medir la eficacia de los árboles obtenidos en cada iteración de ISQL necesitamos diseñar un sistema que nos permita fácilmente cambiar el árbol de clasificación que el agente de la *keepaway* utiliza en cada momento

como aproximador de la función Q. En el caso de utilizar un estimador por acción, necesitamos poder cambiar los *N* estimadores correspondientes a cada una de las *N* acciones que el *Keeper* puede elegir. Los diseños aquí expuestos buscan poder cambiar el estimador en el entorno de pruebas haciendo que este cambio tenga el menor impacto posible.

Modelo ISQL-UNICO de controlador para la evaluación de 1 aproximador

Para probar los aproximadores generados por ISQL que utilizan como entrada la acción, además del estado, se ha diseñado una arquitectura sencilla cuya idea es implementar un controlador para un agente que facilite cambiar el módulo del aproximador por otro distinto. El módulo del aproximador deberá definir una interfaz que servirá para facilitar el acoplamiento del aproximador con el resto de módulos del controlador. La figura 38 muestra el modelo de controlador diseñado y los módulos implicados.

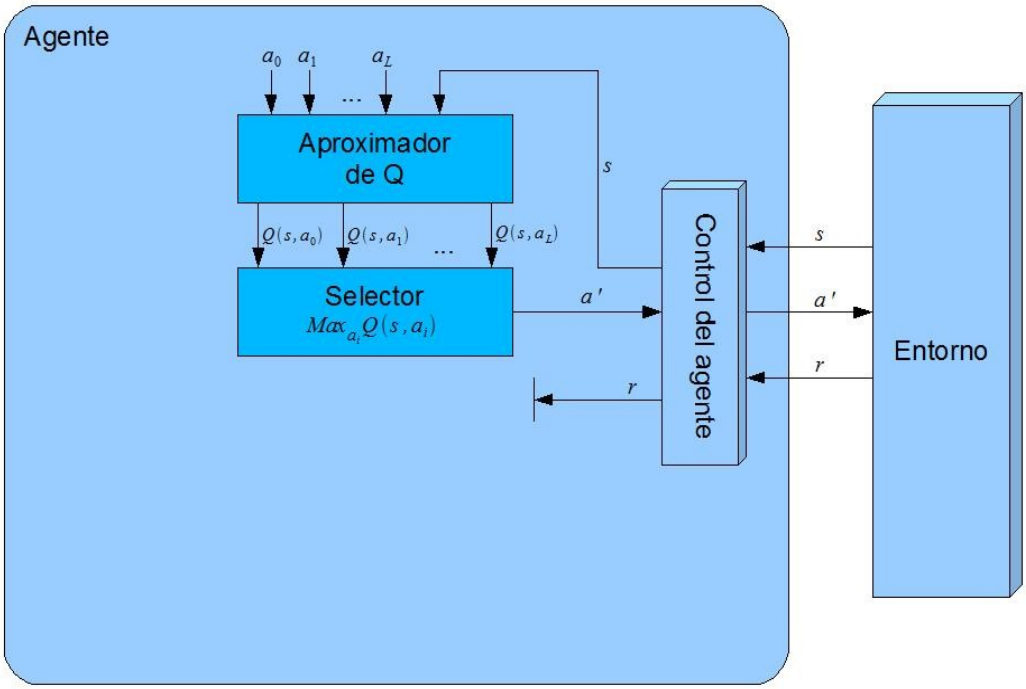

Figura 38: Modelo ISQL-UNICO de controlador para la evaluación de 1 aproximador.

El funcionamiento del modelo es el siguiente:

1. El *entorno* informa al agente del estado, *s*, en el que se encuentra.
2. El agente pasa la información de estado al *controlador*.
3. El controlador pasa al *aproximador* el estado recibido y cada una de las acciones, a_i, posibles.

4. El *aproximador* calcula para cada par estado-acción el beneficio esperado, $Q(s,a_l)$. En el caso de la *keepaway* la salida del *aproximador* corresponde con el tiempo previsto de posesión del balón si se ejecuta la acción en el estado recibido.
5. El *selector* recibe los refuerzos calculados por el *aproximador* e identifica la acción que mejor beneficio reporta.
6. El *selector* informa de la mejor acción , *a'*, al *control* del agente para que sea ejecutada.
7. El *control* ejecuta la acción *a'* y el *entorno* devuelve el estado en el que se encuentra el agente después de ejecutar la acción y el beneficio real obtenido, el cual es desechado.

Con este esquema de funcionamiento el *aproximador* funciona como una caja negra que según el estado y la acción que le proporcionemos, nos devuelve el refuerzo esperado. Este esquema permite cambiar el *aproximador* de manera sencilla, tan solo cambiando la implementación del módulo *aproximador*. La figura 39 adapta el modelo descrito al caso concreto de la *keepaway*.

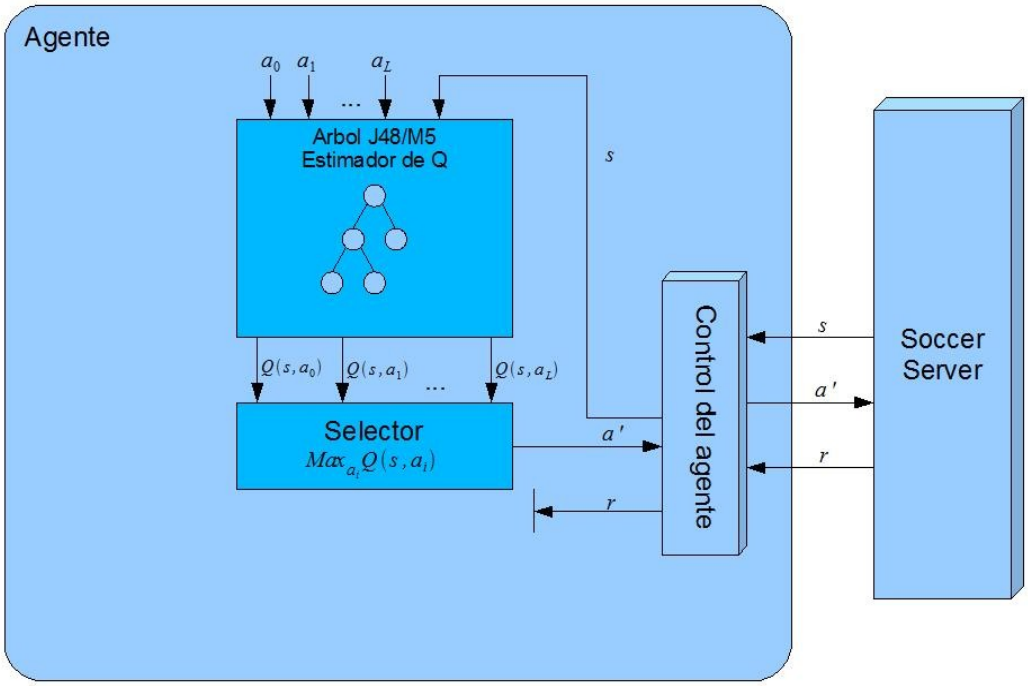

Figura 39: Modelo de controlador con 1 aproximador para *keepaway*

Modelo ISQL-MULTIPLE de controlador para la evaluación con L aproximadores

El siguiente modelo está orientado a la evaluación de la aproximación de la función Q con L aproximadores, los cuales se corresponden con cada una de las L acciones que el agente puede ejecutar. En este caso la entrada del aproximador solo es el estado en el que se encuentra el agente, Ya que la acción ya viene implícita en cada aproximador. Se mantiene el esquema del modelo para 1 aproximador descrito en el apartado anterior con algunas modificaciones que solo afectan al módulo del aproximador. Realmente el cambio reside en el interior del aproximador ya que a la vista del resto de módulos el módulo aproximador mantendría su interfaz. Ahora el aproximador no se limita a implementar la interfaz del módulo aproximador sino que hace de *proxy* o mediador entre los estimadores y el resto de módulos. El módulo aproximador delega los cálculos de los beneficios esperados a cada un de los estimadores por acción. El esquema del modelo se puede ver en la figura 40.

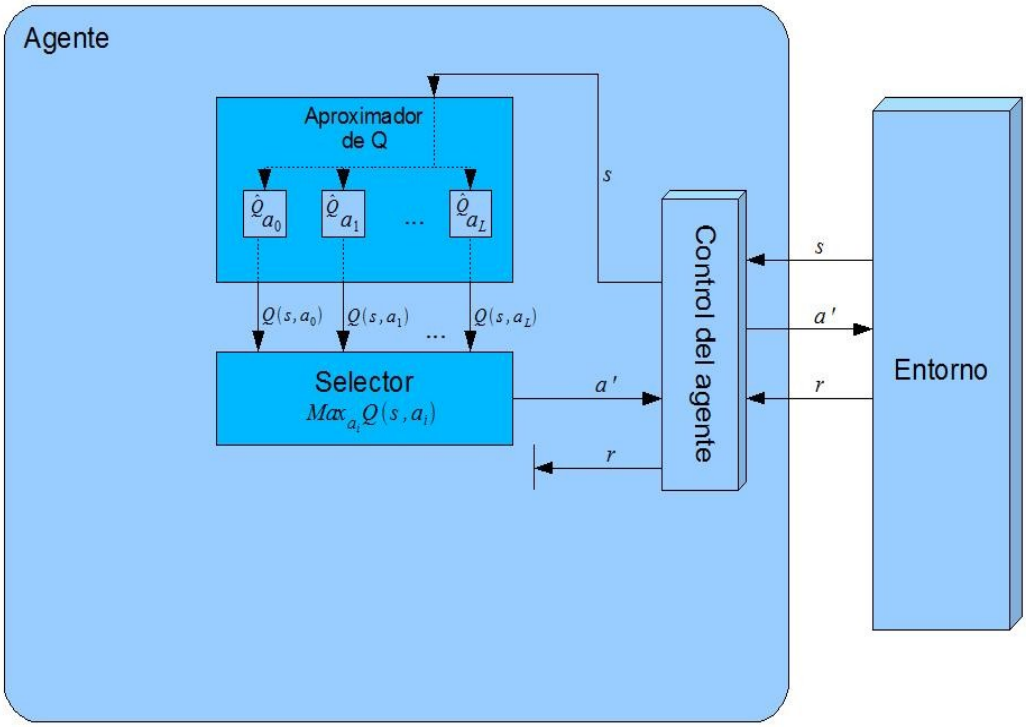

Figura 40: Modelo ISQL-MULTIPLE de controlador con L aproximadores.

3.2.2 Arquitectura de evaluación de MDQL

Como se explicó en el aparatado 3.1, los aproximadores generados en la fase ISQL son utilizados por los agentes que evalúan (ejecutan) la fase MDQL como discretizador del espacio de estados. De forma más concreta, la función de los aproximadores en los modelos que se describen a continuación es la de ser utilizados como indizador de la tabla Q, haciendo corresponder el estado que recibe el agente desde el entorno con una posición de la tabla. Una vez explicada la función que desempeñan los aproximadores en esta fase, pasamos a ver el modelo que permite probar la bondad de la división del espacio de estados ofrecida por los clasificadores obtenidos en la fase ISQL. Se puede observar el modelo MDQL-UNICO para 1 aproximador en la figura 41.

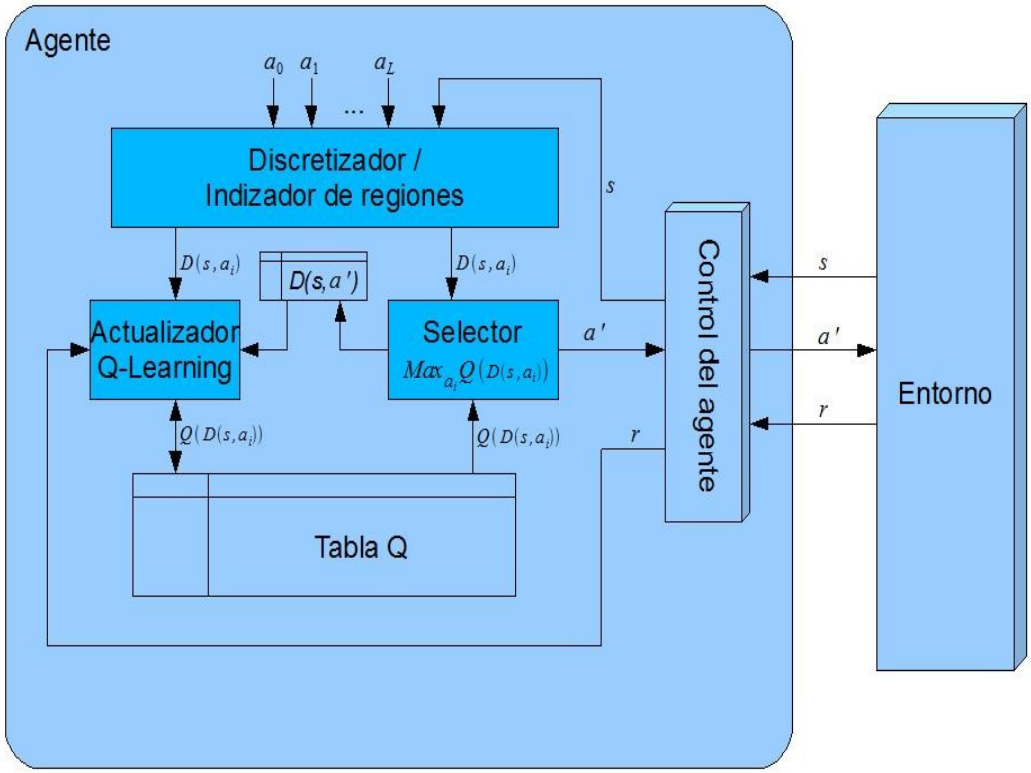

Figura 41: Modelo MDQL-UNICO de controlador con 1 aproximador junto con el algoritmo Q-Learning.

La explicación del modelo es la siguiente:

1. El *entorno* informa al agente del estado, *s*, en el que se encuentra.
2. El controlador del agente pasa al *discretizador/indizador* el estado recibido junto con las acciones posibles a ejecutar, a_i.

3. Por cada acción el *discretizador* devuelve un índice, $D(s, a_i)$, que indica que posición de la tabla Q se debe consultar. Estos índices son recibidos por el *selector*.
4. El selector consulta las posiciones de la tabla Q indicadas por el *discretizador* y localiza la que reporta mayor beneficio, devolviendo la acción correspondiente que será ejecutada, *a'*. También se guarda en memoria la posición de la tabla Q que corresponde con el valor máximo de los consultados $D(s, a')$.
5. La acción devuelta por el selector se ejecuta y el entorno devuelve el refuerzo real obtenido, *r*, y el estado al que se ha transitado.
6. El refuerzo recibido es utilizado por el actualizador *Q-Learning* para modificar la casilla $D(s, a')$ de tabla Q según la formula de actualización del algoritmo *Q-Learning*.

El modelo con MDQL-MULTIPLE con 1 aproximador por acción es muy similar al que utiliza un solo aproximador para todas las acciones. La principal diferencia es que la obtención del índice se delega a los L aproximadores existentes. En concreto se consulta al aproximador que corresponda con la acción del par estado-acción de entrada. El modelo se puede ver en la figura 42.

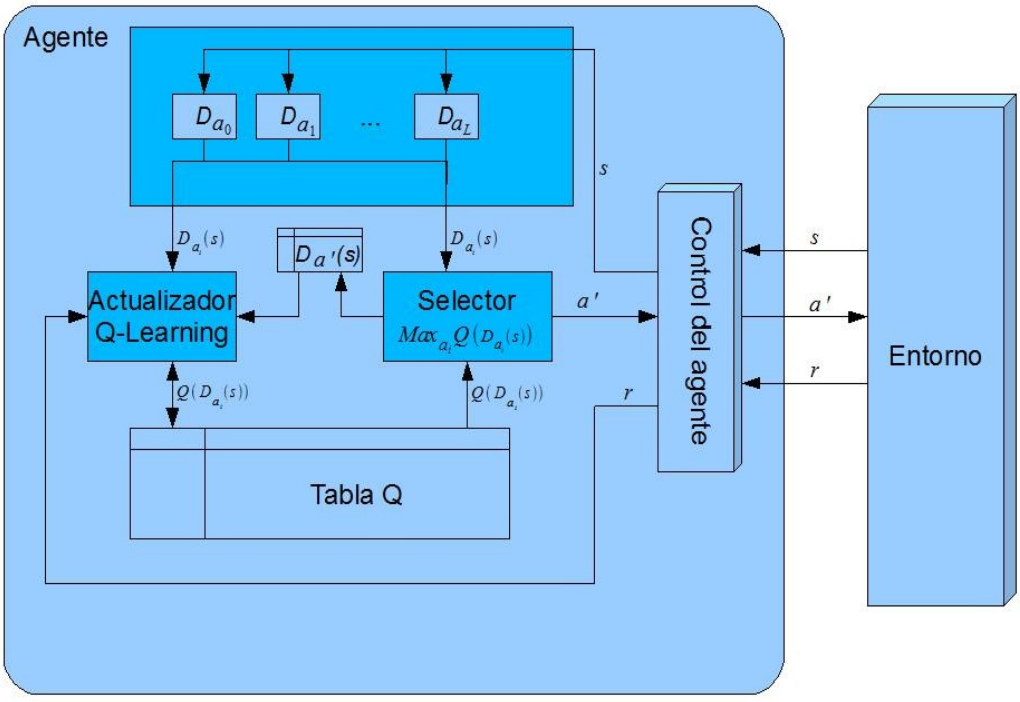

Figura 42: Modelo MDQL-MULTIPLE de controlador con L aproximadores junto con el algoritmo Q-Learning.

A los anteriores modelos expuestos podemos cambiarles el módulo de actualización *Q-Learning* por cualquier otro método de actualización de la tabla Q como puede ser *Q(λ)*, *Sarsa* o *Sarsa(λ)*. En el actual proyecto se han implementado agentes que utilizan *Q-Learning* y *Q(λ)*. Además, se ha añadido a los modelos un nuevo módulo que permite al agente utilizar una acción diferente a la mejor, según la estrategia de exploración deseada. En este proyecto se ha implementado la estrategia $\epsilon - greedy$ integrada en los anteriores modelos. En la figura 43 se puede observar a modo de ejemplo como queda integrado el módulo $\epsilon - greedy$ en el modelo MDQL-MULTIPLE.

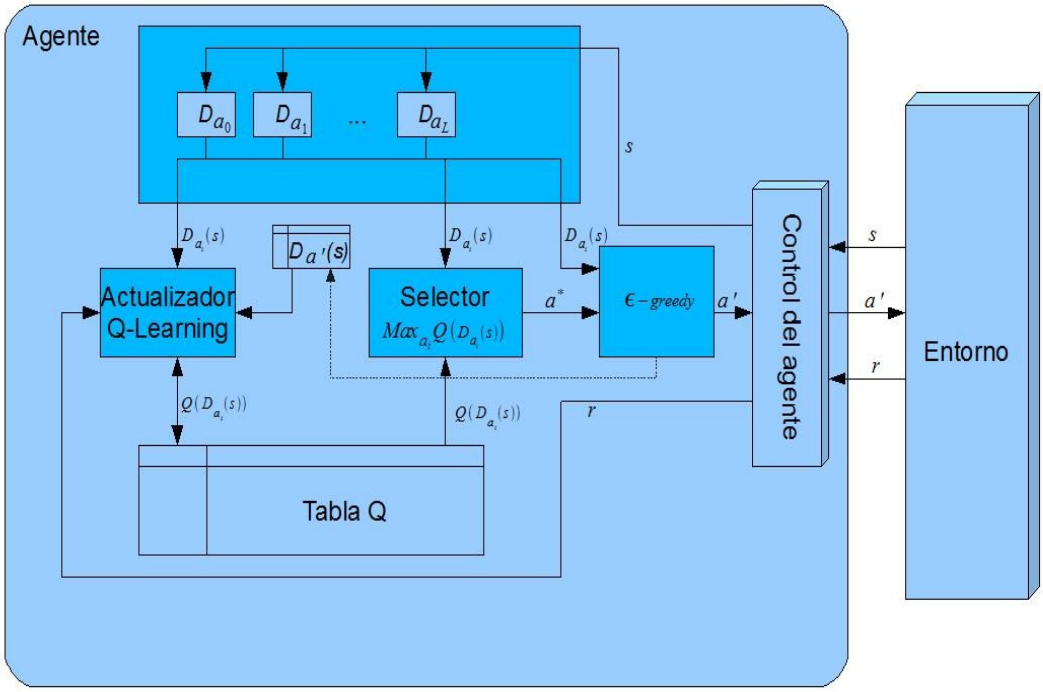

Figura 43: Modelo MDQL-MULTIPLE junto con el algoritmo Q-Learning y la estrategia de exploración epsylón greedy.

4 Evaluación

El apartado de evaluación recoge la experimentación realizada con la *keepaway* para probar el método *Two Steps Reinforcement Learning* (2SRL). El algoritmo utiliza un gran número de variables cuyo valor inicial debe ser configurado. Además podemos distinguir entre la configuración de la fase *Iterative Smooth Q-Learning* (ISQL) y la configuración de la fase *Multiple Discretization Q-Learning* (MDQL).

Para simplificar la experimentación hemos fijado los parámetros del juego de la *keepaway* en los siguientes datos:

- El número de *keepers* siempre es 3.
- El número de *takers* siempre es 2.
- Las dimensiones del terreno de juego son 25x25.
- No hay restricciones de visión. Los jugadores pueden captar el entorno en 360 grados.

Al margen del dominio de prueba, los parámetros de configuración propios del algoritmo 2SRL siguen produciendo infinidad de combinaciones. Con respecto a la fase ISQL los parámetros que podemos configurar son los siguientes:

- Utilización de uno o varios aproximadores (uno por acción) en el algoritmo.
- Utilización de la función de actualización de *Q-Learning* para dominios estocásticos o deterministas (determinado por los valores de α y Υ).
- Algoritmo de clasificación o regresión (en este caso J48 o M5P).
- Parámetros del algoritmo de clasificación o regresión (factor de poda, número mínimo de instancias por hoja, etc).
- En caso de utilizar J48, discretización del espacio de recompensas (número de clases, rangos de discretización, etc).

Del mismo modo, los parámetros que se pueden configurar en la fase MDQL son los siguientes:

- Parámetros α y Υ de la función de actualización de *Q-Learning*.
- Factor de crecimiento de ϵ por episodio en la política de exploración $\epsilon - greedy$.
- Modo de inicialización de la tabla Q. Puede inicializarse con todos los valores a cero o inicializar la tabla Q con los valores dados por la política calculada en la fase ISQL.
- Uso de trazas de elegibilidad (valor de λ).

En este apartado se analizan las configuraciones que se han considerado de mayor relevancia, tanto por su propuesta como por sus resultados. En un primer apartado se describe el formato de los experimentos para clarificar la comprensión de los mismos. Después se pasa a describir los resultados con el formato especificado. La experimentación se ha dividido en dos grupos: la fase ISQL y la fase MDQL. En ambas fases se distingue entre experimentos realizados con J48 y M5P como algoritmo de aproximación.

4.1 Descripción de los experimentos

En esta sección se dan las pautas necesarias para interpretar de forma correcta y rápida los experimentos de las secciones 4.2 y 4.3. Se define un formato común a todos ellos y se realiza una descripción de los elementos que los componen.

Los experimentos de la fase ISQL ocupan 2 páginas exactamente. Cada página contendrá unos elementos fijos cuya distribución se puede observar en la figura 44:

- La primera página de cada experimento contiene una descripción textual del mismo, donde se comentan los resultados obtenidos. Además en esta primera página también se incluye la tabla de configuración de las variables del experimento.

- La segunda página incluye 3 gráficas: una de la evolución del error, otra de la evolución del número de regiones y una de la evolución de la duración media de los episodios por cada iteración de ISQL.

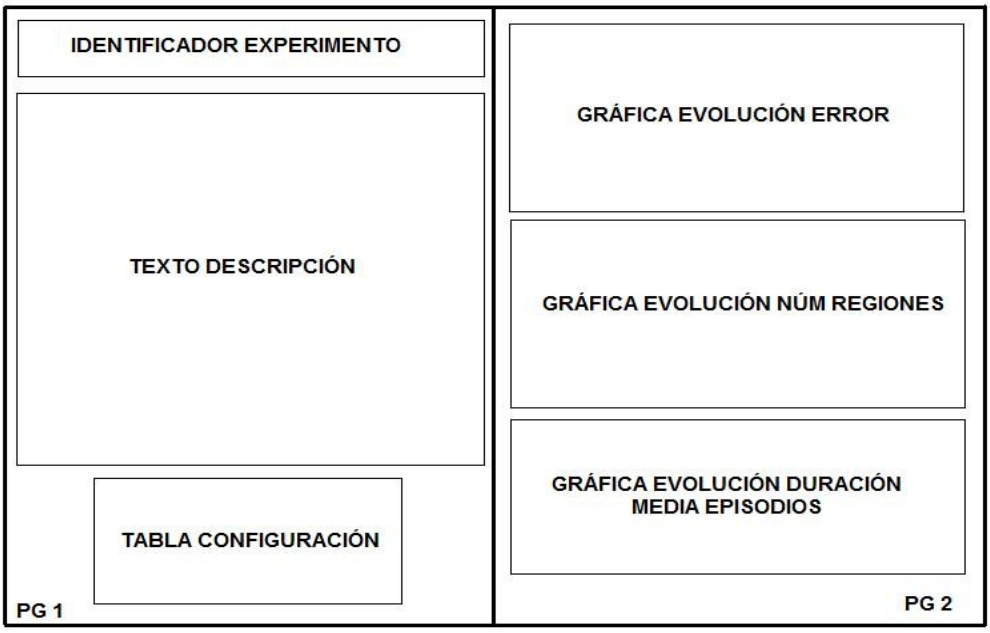

Figura 44: Formato de los experimentos de la fase ISQL

La tabla de configuración indica los valores de las variables utilizadas en el experimento. En la tabla 7 se puede ver un ejemplo de tabla de configuración. Los valores que no apliquen en cada caso estarán marcados con el valor *X*. La explicación de cada una de las variables es la siguiente:

- **α:** Parámetro de la ecuación de actualización de *Q-learning* para dominios estocásticos. Define la importancia que se da a las nuevas actualizaciones con respecto a valores anteriores. Este valor está siempre entre 0 y 1. Si este parámetro toma valor 1 se obtiene la ecuación de actualización para dominios deterministas.
- **γ:** Parámetro de descuento del criterio de optimalidad del horizonte infinito descontado.
- **C:** Factor de poda de los árboles del algoritmo J48 de WEKA.
- **M:** Número mínimo de instancias por hoja de los árboles. Este parámetro es común al algoritmo J48 y M5P de WEKA.
- **R:** Parámetro de configuración del algoritmo M5P. Si se configura como regresión el resultado es un árbol de regresión cuyas hojas contienen el valor medio de las instancias de cada hoja. Si se configura como modelos cada hoja del árbol contendrá un modelo de regresión que suaviza la predicción del valor de cada instancia.
- **Función aproximada:** Cuando este parámetro especifica un número de clases significa que estamos usando el algoritmo de clasificación J48 para aproximar la función Q y el número de clases indica el número de niveles en el que se ha discretizado el espacio de recompensas. Si este parámetro especifica el valor "continua" significa que hemos utilizado el algoritmo de regresión M5P para aproximar la función Q.
- **Aproximador:** Cuando el valor de esta variable es "Único" indica que se utiliza un solo aproximador de la función Q para todas las acciones, incluyendo la acción como un parámetro más de entrada del aproximador. Cuando el valor es "Múltiple" indica que la aproximación de Q se realiza con un aproximador para cada acción. Como todos los experimento están realizados sobre un juego de la keepaway 3vs2, lo que significa que siempre que se use un aproximador múltiple se utilizarán 3 aproximadores: uno para la acción hold, otro para pass keeper 1 y otro para pass keeper 2.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	0,25
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	X
Función aproximada (discreta [Nº clases]/continua)	88 clases
Aproximador (Único/Múltiple)	Único

Tabla 7: Tabla de configuración de ejemplo.

Con respecto a la variable **Función aproximada** se han utilizado dos discretizaciones distintas en el caso de utilizar J48 + discretización como aproximación de Q:

- **Discretización del espacio de recompensas con 12 clases**

Se realiza una discretización gruesa del espacio de recompensas, generando contenedores que abarcan grandes rangos de valores, generando 12 contenedores que resumen el espacio de recompensas. La figura 45 muestra la discretización realizada en el Explorer de Weka.

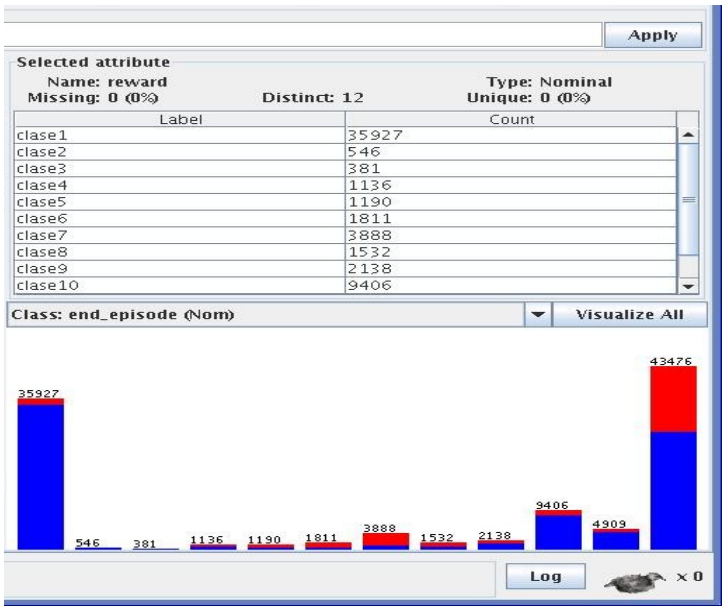

Figura 45: Discretización de reward con 12 clases.

El criterio de discretización intenta generar conjuntos uniformes en el número de instancias dentro de las posibilidades de las instancias de las que se dispone, para así tener conjuntos representativos de cada clase y facilitar la clasificación. Los rangos de la discretización generada se muestran en la tabla 8.

Inicio rango	Fin rango	clase
-50	2.320513	clase1
2.320513	3.641026	clase2
3.641026	4.961538	clase3
4.961538	6.282051	clase4
6.282051	7.602564	clase5
7.602564	8.923077	clase6
8.923077	10.24359	clase7
10.24359	11.564103	clase8
11.564103	12.884615	clase9
12.884615	14.205128	clase10
14.205128	15.525641	clase11
15.525641	200	clase12

Tabla 8: Rango de discretizaciones para 12 clases.

■ Discretización del espacio de recompensas con 88 clases

Se realiza una discretización fina del espacio de recompensas, generando contenedores que abarcan pequeños rangos de valores, generando 88 contenedores que resumen el espacio de recompensas. La figura 46 muestra la discretización realizada en el Explorer de Weka.

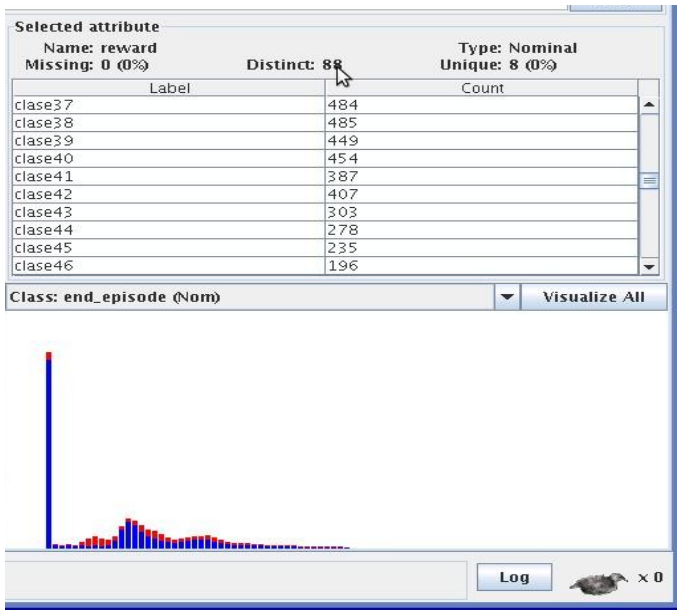

Figura 46: Discretización de reward con 88 clases.

El criterio de discretización intenta generar conjuntos uniformes en el número de instancias dentro de las posibilidades de las instancias de la que se dispone, para así tener conjuntos representativos de cada clase y facilitar la clasificación. Los rangos de la discretización generada se muestran en la tabla 9.

Inicio rango	Fin rango	clase	Inicio rango	Fin rango	clase	Inicio rango	Fin rango	clase
-50	1.5	clase1	30.5	31.5	clase30	59.5	60.5	clase59
1.5	2.5	clase2	31.5	32.5	clase31	60.5	61.5	clase60
2.5	3.5	clase3	32.5	33.5	clase32	61.5	62.5	clase61
3.5	5.5	clase4	33.5	34.5	clase33	62.5	63.5	clase62
5.5	6.5	clase5	34.5	35.5	clase34	63.5	64.5	clase63
6.5	7.5	clase6	35.5	36.5	clase35	64.5	65.5	clase64
7.5	8.5	clase7	36.5	37.5	clase36	65.5	66.5	clase65
8.5	9.5	clase8	37.5	38.5	clase37	66.5	67.5	clase66
9.5	10.5	clase9	38.5	39.5	clase38	67.5	68.5	clase67
10.5	11.5	clase10	39.5	40.5	clase39	68.5	69.5	clase68
11.5	12.5	clase11	40.5	41.5	clase40	69.5	70.5	clase69
12.5	13.5	clase12	41.5	42.5	clase41	70.5	71.5	clase70
13.5	14.5	clase13	42.5	43.5	clase42	71.5	72.5	clase71
14.5	15.5	clase14	43.5	44.5	clase43	72.5	73.5	clase72
15.5	16.5	clase15	44.5	45.5	clase44	73.5	74.5	clase73
16.5	17.5	clase16	45.5	46.5	clase45	74.5	75.5	clase74
17.5	18.5	clase17	46.5	47.5	clase46	75.5	76.5	clase75
18.5	19.5	clase18	47.5	48.5	clase47	76.5	77.5	clase76
19.5	20.5	clase19	48.5	49.5	clase48	77.5	79	clase77
20.5	21.5	clase20	49.5	50.5	clase49	79	81.5	clase78
21.5	22.5	clase21	50.5	51.5	clase50	81.5	83.5	clase79
22.5	23.5	clase22	51.5	52.5	clase51	83.5	84.5	clase80
23.5	24.5	clase23	52.5	53.5	clase52	84.5	85.5	clase81
24.5	25.5	clase24	53.5	54.5	clase53	85.5	86.5	clase82
25.5	26.5	clase25	54.5	55.5	clase54	86.5	88	clase83
26.5	27.5	clase26	55.5	56.5	clase55	88	89.5	clase84
27.5	28.5	clase27	56.5	57.5	clase56	89.5	90.5	clase85
28.5	29.5	clase28	57.5	58.5	clase57	90.5	93.5	clase86
29.5	30.5	clase29	58.5	59.5	clase58	93.5	100	clase87
						100	200	clase88

Tabla 9: Rango de discretizaciones para 88 clases.

A continuación se explica la interpretación de las gráficas que se incluyen en cada experimento de ISQL:

- *Gráfica de la evolución del error*: esta gráfica representa la evolución de la distancia entre los aproximadores por cada iteración. El eje *X* representa la iteración de ISQL y el eje Y representa la distancia entre el aproximador de la iteración *N* y el de la *N-1.* Como medida de distancia se utiliza la distancia euclídea entre el vector de recompensas obtenidas en la iteración *N* y el de la *N-1.* Cuando utilizamos un aproximador único la gráfica tendrá solo una curva de evolución. En cambio, si utilizamos un aproximador múltiple la gráfica mostrará una curva de evolución por cada acción.
- *Gráfica de la evolución del número de regiones*: Muestra la evolución del número de hojas del estimador generado en cada iteración. El eje *X* representa la iteración de ISQL y el eje *Y* representa el número de hojas o regiones del aproximador en cada iteración. Cuando utilizamos un aproximador único la gráfica tendrá solo una curva de evolución. En cambio, si utilizamos un aproximador múltiple la gráfica mostrará una curva de evolución por cada acción.
- *Gráfica de la evolución de la duración media de los episodios*: Muestra la evolución de la duración media de los episodios jugados utilizando las políticas generadas a partir de los aproximadores de cada iteración de ISQL. El eje *X* es la iteración de ISQL y el eje *Y* representa la duración media de los episodios la donde los *keepers* utilizan el aproximador de cada iteración para calcular la política de acción.

A cada experimento ISQL le corresponde directamente un experimento MDQL. De esta forma queda cubierta la experimentación con 2SRL mediante la evaluación de sus dos fases. Los experimentos de la fase MDQL ocupan 3 páginas exactamente. Cada página contendrá unos elementos fijos cuya distribución se puede observar en la figura 47:

- La primera página de cada experimento contiene una descripción textual del mismo, donde se comentan los resultados obtenidos.

- La segunda página incluye las gráficas de evolución del aprendizaje de los *keepers* en MDQL correspondientes a las iteraciones más significativas de ISQL. Según el número de gráficas añadidas esta sección puede saltar a la tercera página.

- La tercera página de cada experimento MDQL recoge una tabla resumen de las duraciones medias de los episodios obtenidas con las políticas de ISQL y MDQL para las iteraciones más significativas. También resume información adicional sobre el número de regiones de los aproximadores en las iteraciones estudiadas. Esta tercera página también recoge una gráfica comparativa entre la evolución de la políticas obtenidas con ISQL y MDQL.

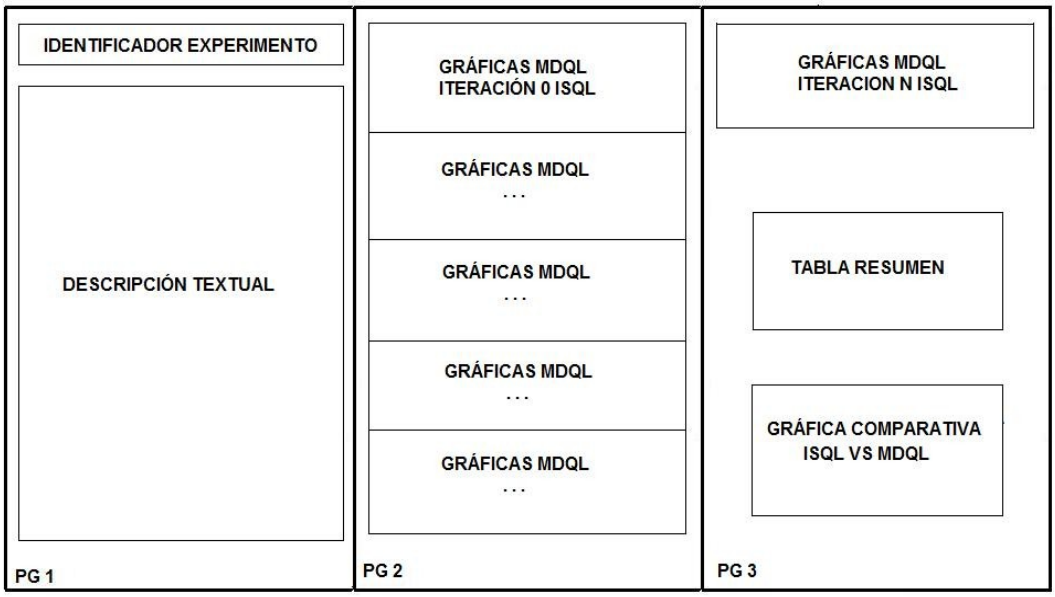

Figura 47: Formato de los experimentos de la fase ISQL

Las gráficas de la segunda página de cada experimento son como la que se muestra en la figura 48. Cada una de estas gráficas muestra la evolución de la duración media de los episodios de una simulación en la que se ha utilizado *Q-learning* junto con la discretización del espacio de estados obtenida a partir del aproximador de la iteración indicada de ISQL. Los parámetros de *Q-Learning* utilizados para todas las gráficas son $\alpha = 0{,}125$ y $\Upsilon = 1$. Estos valores han sido escogidos observando los valores utilizados en anteriores trabajos, los cuales dieron buenos resultados [García et al, 2007]. La estrategia de exploración es $\epsilon - greedy$ con un incremento en ϵ de 0,0001 por cada episodio transcurrido. Justo encima de cada gráfica se especifica como se ha inicializado la tabla Q. Se ha realizado experimentación inicializando las casillas de la tabla Q a cero (Q = 0) e inicializando la tabla Q con la política estimada por el aproximador de cada iteración ($Q = Q^{iter}$), que no es más que el valor con el que estaba etiquetada cada hoja del árbol aproximador. Además, todas estas gráficas están agrupadas por la iteración de la que se obtuvo la discretización del espacio de estados. En el eje X se representa el tiempo de entrenamiento transcurrido. En el eje Y se representa la duración media que obtienen los episodios en el transcurso de la simulación. Un crecimiento de la media significa una mejora en la política de acción de los *keepers*.

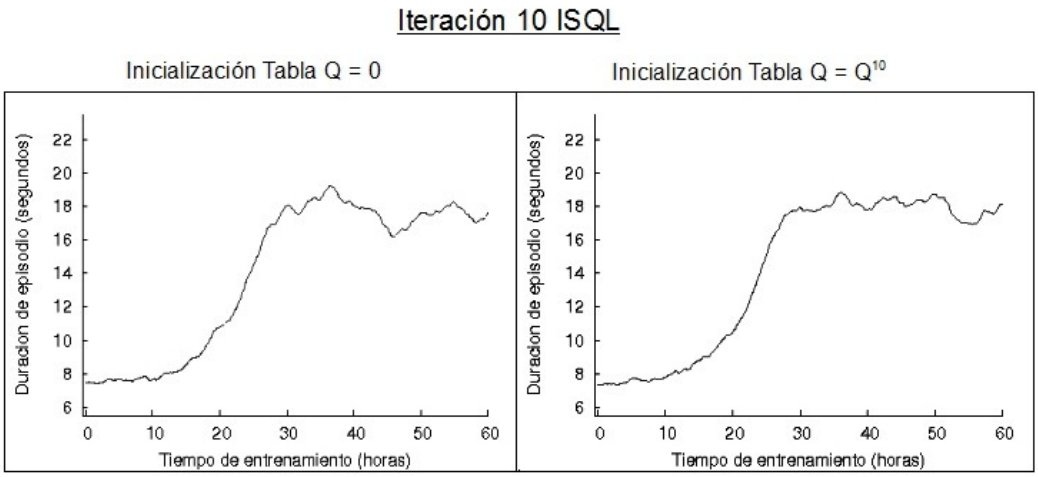

Figura 48: Ejemplo de gráficas de simulación MDQL
(utilizando Q-Learning + discretización iteración 10 de ISQL)

La información recogida en la tabla resumen se puede observar en el ejemplo tabla 10. La fila $\bar{x}_{ISQL}$ muestra la duración media de los episodios jugados en cada iteración con la política aprendida en la fase ISQL. La fila $\bar{x}_{MDQL-Q=0}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con ceros. La fila $\bar{x}_{MDQL-Q^{iter}}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con la política obtenida en la iteración correspondiente de ISQL. Por último, la última fila muestra el número de regiones en las que el estimador de cada iteración divide el espacio de estados. Dependiendo del experimento, se ha optado por prescindir de alguna de las filas por no considerarse relevante para ese experimento.

Iteración	0	10	60	80
$\bar{x}_{ISQL}$	8,22	10,88	10,81	12,22
$\bar{x}_{MDQL-Q^0}$	11,56	18,10	19,13	17,30
$\bar{x}_{MDQL-Q^{iter}}$	11,58	17,98	19,00	18,50
Nº regiones	79 (1+27+51)*	173 (44+60+69)*	553 (191+197+165)*	566 (202+187+177)*
* Desglose por acción: (hold + pass k1 + pass k2)				

Tabla 10: Tabla resumen de resultados MDQL de ejemplo.

En la gráfica comparativa se pueden observar gráficamente los datos de la tabla resumen. La gráfica muestra la comparativa entre los resultados de las distintas fases de 2SRL y las distintas inicializaciones de Q. Un ejemplo de gráfica comparativa puede verse en la figura 49.

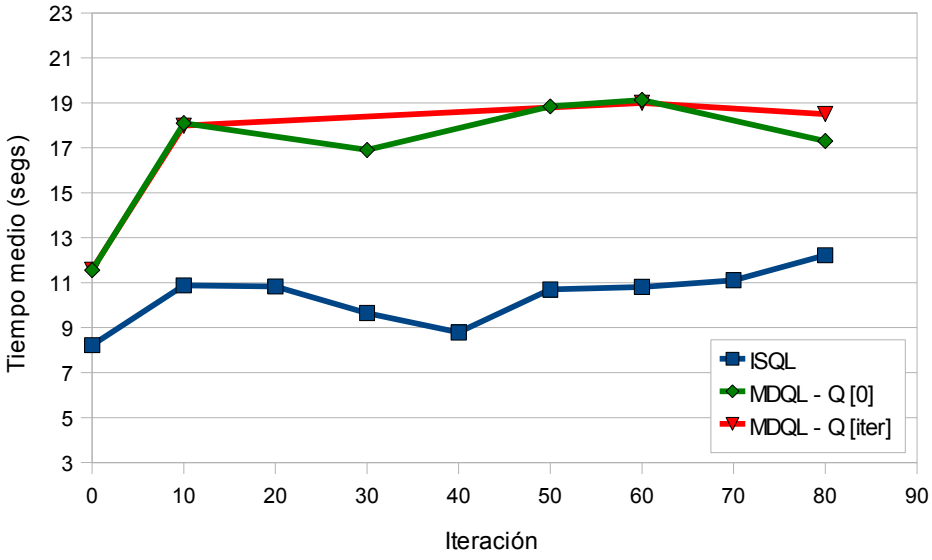

Figura 49: Ejemplo de gráfica comparativa ISQL vs MDQL.

Para concluir este apartado se muestra la tabla 11, la cuál resume todos los experimentos tratados en los apartados 4.2 y 4.3. Además de indicar la configuración de los experimentos ISQL, la última columna indica la correspondencia entre los experimentos ISQL y su respectiva fase MDQL.

Experimento	Configuración							MDQL
Identificador	α	γ	C	M	R	F. Aproximada	Aproximador	Identificador
ISQL-1	0,125	1	0,25	100	X	88 clases	Único	MDQL-1
ISQL-2	0,125	1	0,25	100	X	12 clases	Único	MDQL-2
ISQL3	1	1	0,25	100	X	12 clases	Único	MDQL-3
ISQL-4	0,125	1	0,25	100	X	88 clases	Múltiple	MDQL-4
ISQL-5	0,125	1	0,25	100	X	12 clases	Múltiple	MDQL-5
ISQL-6	0,125	1	X	100	Modelos	Continua	Único	MDQL-6
ISQL-7	0,125	1	X	100	Regresión	Continua	Único	MDQL-7
ISQL-8	0,125	1	X	4	Modelos	Continua	Múltiple	MDQL-8
ISQL-9	0,125	1	X	100	Modelos	Continua	Múltiple	MDQL-9
ISQL-10	0,125	1	X	50	Regresión	Continua	Múltiple	MDQL-10
ISQL-11	0,125	1	X	100	Regresión	Continua	Múltiple	MDQL-11

Tabla 11: Resumen de la experimentación.

4.2 Experimentos de la fase ISQL

Para la fase ISQL diferenciamos dos tipos de experimentos. En primer lugar se comentan los experimentos en los ha sido utilizado J48 como algoritmo para el cálculo de los modelos de clasificación. En segundo lugar se tratarán los experimentos que utilizan M5P como algoritmo para genera modelos de regresión.

4.2.1 Experimentación con J48

Los experimentos con J48 necesitan de un tratamiento previo del espacio de recompensas. Necesitamos discretizar el espacio de recompensas para que el algoritmo J48 pueda realizar el proceso de clasificación, ya que no acepta clases continuas. Como se explica en el apartado 4.1 se ha realizado una discretización de 12 niveles y otra de 88. Para solucionar este problema se ha experimentado con dos discretizaciones distintas que se comentan a continuación.

Experimento ISQL-1

Como se puede observar en la figura ISQL1-1 correspondiente a la evolución del error o distancia, la gráfica evoluciona favorablemente reduciendo la distancia en cada iteración. Esto indica que existe convergencia y evolución en el algoritmo. Alrededor de la iteración 70 la distancia parece estabilizarse en torno a 150. La figura ISQL1-2 muestra la evolución del número de hojas del estimador generado en cada iteración. Se observa, al igual que en la gráfica de distancia, una convergencia que desemboca al crecimiento del número de hojas hasta que se estabiliza en torno a la iteración 70 en un valor alrededor de 550 hojas.

En la figura ISQL1-3 podemos observar la evolución de la eficacia del estimador obtenido en cada iteración para determinar la política a seguir. La evolución es positiva y el estimador, en cada iteración, es más útil para conseguir una política más optima. Observamos como el tiempo medio de los episodios jugados va mejorando. La mejor política conseguida se ha obtenido en la iteración 80 y su duración media es de 11.63 segundos.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	0,25
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	X
Función aproximada (discreta [Nº clases]/continua)	88 clases
Aproximador (único/múltiple)	Único

Tabla 12: Configuración del experimento ISQL1.

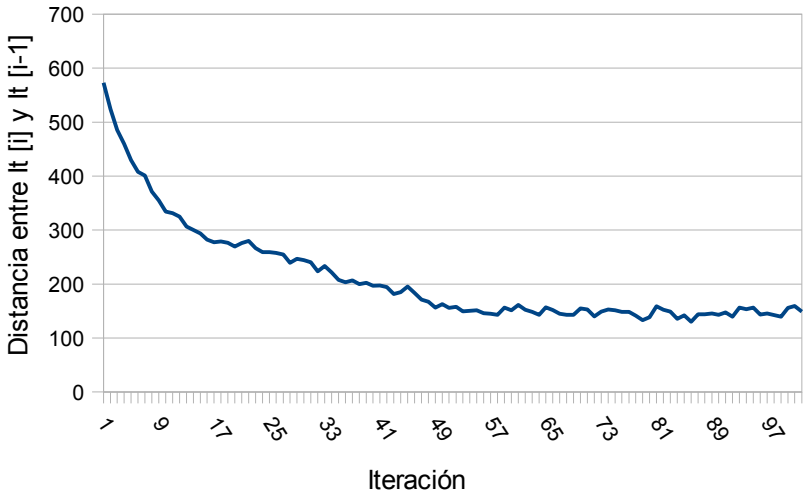

Figura ISQL1-1: Evolución de la distancia entre iteraciones.

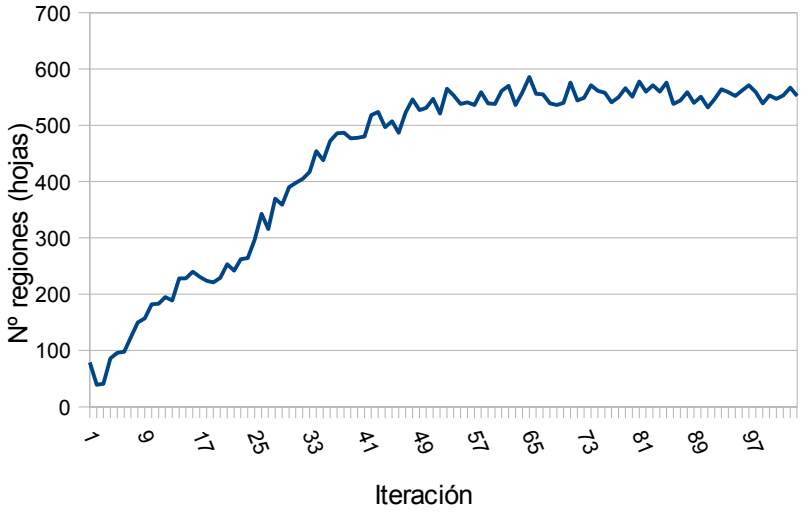

Figura ISQL1-2: Evolución del número de hojas del estimador.

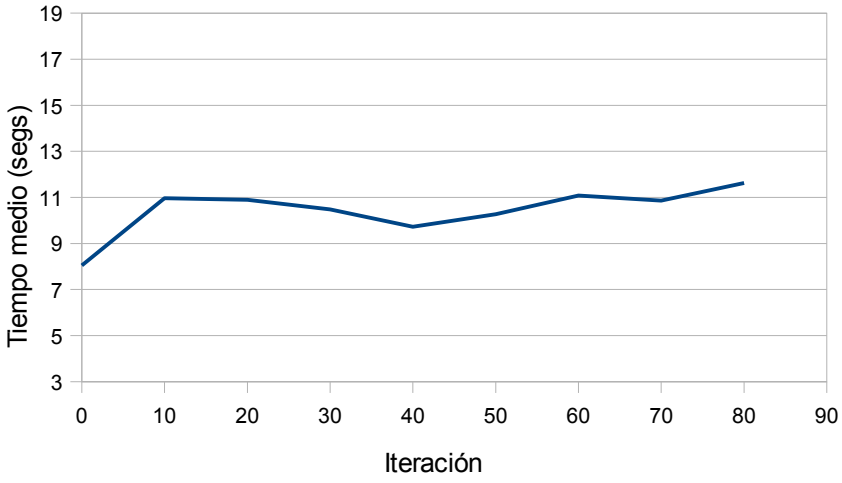

Figura ISQL1-3: Evolución de la duración media por iteración.

Experimento ISQL-2

Los valores de los parámetros son los mismos que los del experimento ISQL1, excepto por la discretización. En este caso utilizamos la discretización de 12 clases. Como se puede observar en la figura ISQL2-1 la gráfica evoluciona reduciendo la distancia en cada iteración de manera más rápida y brusca que cuando utilizábamos la discretización de 88 clases en el experimento ISQL1. El algoritmo converge hasta que en la iteración 52 deja de evolucionar y la distancia entre iteraciones es cero. La figura ISQL2-2 muestra la evolución del número hojas del estimador generado en cada iteración. La convergencia es rápida y el número de regiones o hojas que genera el estimador es menor que las generadas en el experimento ISQL1. Pasamos de estar entre 500 y 600 regiones, a estar rondando entorno a 250. En la figura ISQL2-3 podemos observar la evolución de la eficacia del estimador obtenido en cada iteración para determinar la política a seguir. Se produce una evolución más rápida que en el experimento ISQL1. El pico más alto lo consigue ISQL1 con 11.63 seg, frente a 11.33 seg conseguidos por ISQL2.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	0,25
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	X
Función aproximada (discreta [Nº clases]/continua)	12 clases
Aproximador (único/múltiple)	Único

Tabla 13: Configuración del experimento ISQL2.

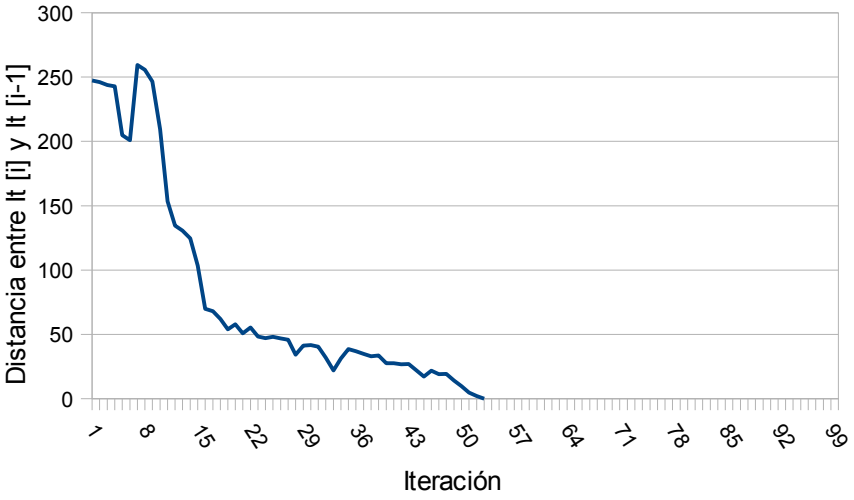

Figura ISQL2-1: Evolución de la distancia entre iteraciones.

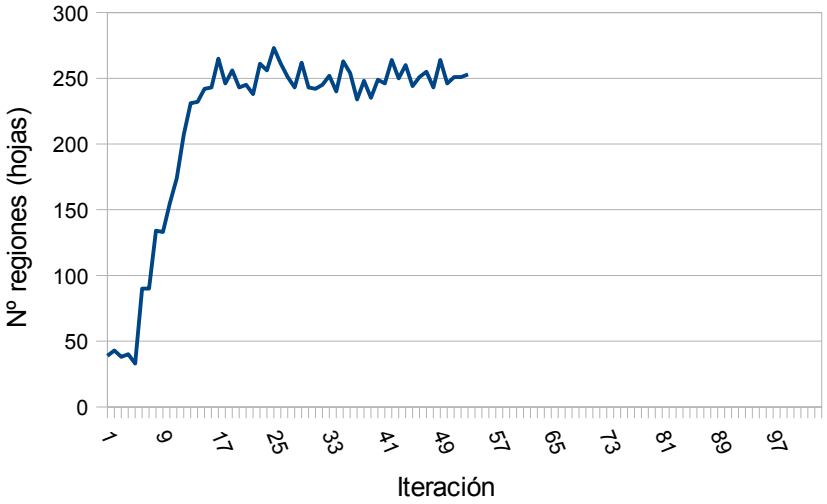

Figura ISQL2-2: Evolución del número de hojas del estimador.

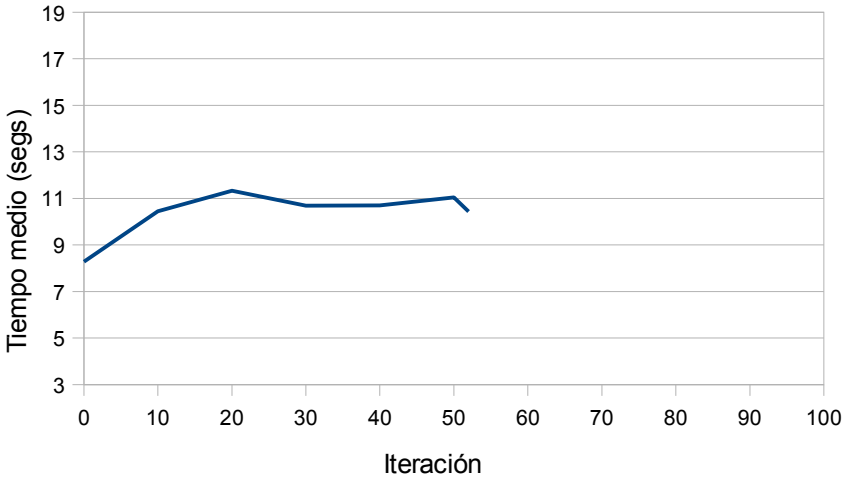

Figura ISQL2-3: Evolución de la duración media por iteración.

Experimento ISQL-3

En este experimento hemos utilizado el valor de *α=1*. Esto equivale a utilizar la función de actualización para dominios deterministas de *Q-Learning*. La figura ISQL3-1 representa la evolución de la distancia entre el aproximador de la iteración N y la N-1. Como se puede observar en la figura ISQL3-1 la distancia disminuye de forma brusca en las primeras 5 iteraciones y después se estanca alrededor del valor de distancia 1000. La figura ISQL3-2 muestra la evolución del número hojas del estimador generado en cada iteración. De la misma forma que la distancia, el número de regiones o hojas aumenta bruscamente en las primeras iteraciones y se estanca alrededor del valor 200 con oscilaciones. En la figura ISQL3-3 podemos observar la evolución de la eficacia del estimador obtenido en cada iteración para determinar la política a seguir. La políticas obtenidas empeoran en cada iteración y no convergen hacia una política optima. Aunque las gráficas ISQL3-1 y ISQL3-2 sugieren una convergencia, esta convergencia no va encaminada a conseguir un estimador refinado, como se ha podido comprobar en el simulador a la vista de los resultados de la gráfica ISQL3-3. Como se puede comprobar comparando estos resultados con los de el experimento ISQL-2, bajo las mismas condiciones de configuración, la ecuación estocástica de actualización funciona mejor en este dominio que la determinista.

Variables	Valor
α	1
γ	1
C (Factor de poda)	0,25
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	X
Función aproximada (discreta [Nº clases]/continua)	12 clases
Aproximador (único/múltiple)	Único

Tabla 14: Configuración del experimento ISQL3.

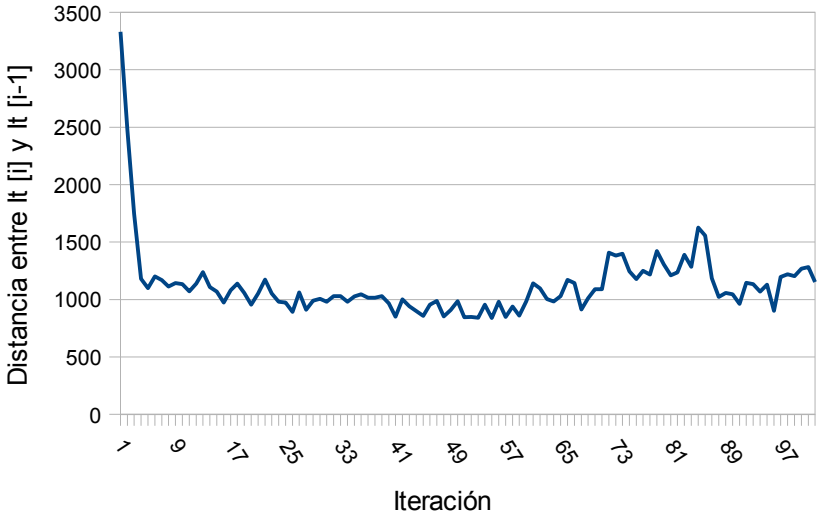

Figura ISQL3-1: Evolución de la distancia entre iteraciones.

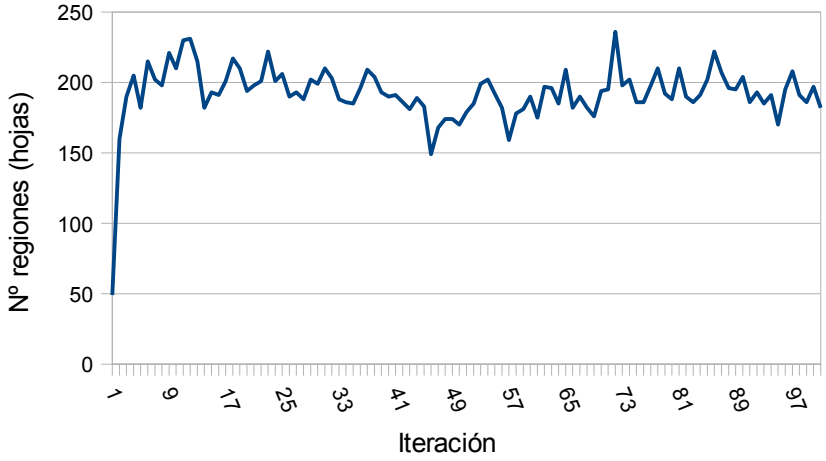

Figura ISQL3-2: Evolución del número de hojas del estimador.

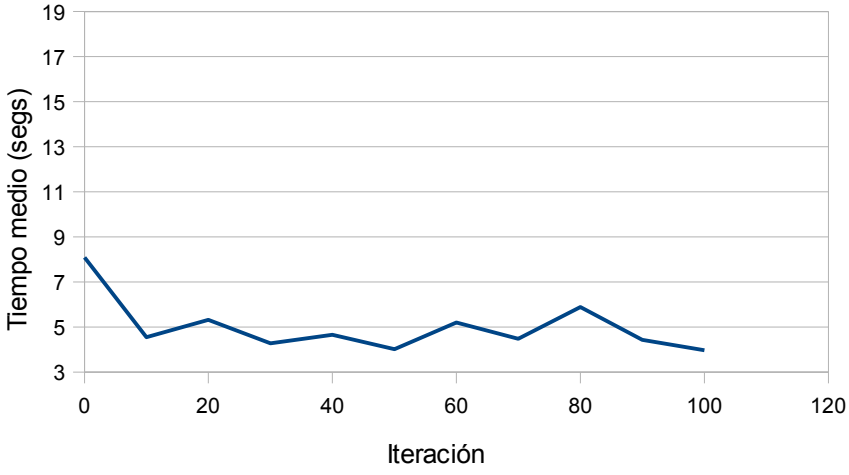

Figura ISQL3-3: Evolución de la duración media por iteración.

Experimento ISQL-4

Este experimento es equivalente al experimento ISQL-1 con la diferencia de que esta vez utilizamos un estimador para cada acción, es decir, un aproximador múltiple. La figura ISQL4-1 representa la evolución de la distancia entre el aproximador de la iteración N y la N-1. Cada una de las 3 curvas se corresponden con la evolución de cada uno de los 3 estimadores para las 3 acciones: *Hold*, *Pass Keeper 1* y *Pass Keeper 2*. Las distancias disminuyen suavemente de manera que se consigue una convergencia. Observamos que para la acción *hold* primero se produce un crecimiento de la distancia y en la iteración 10 comienza a disminuir. La figura ISQL4-2 muestra la evolución del número hojas de cada estimador. Se puede observar que la evolución del número de regiones de los tres estimadores es positiva, generándose en cada iteración más regiones hasta que se estabiliza. En la figura ISQL4-3 podemos observar la evolución de la eficacia del estimador obtenido en cada iteración para determinar la política a seguir. Se observa una buena evolución de la política obtenida a partir de los estimadores. La mejora de la política alcanza su punto más alto en la iteración 80 con una media de duración de cada episodio de 12,22 segundos.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	0,25
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	X
Función aproximada (discreta [Nº clases]/continua)	88 clases
Aproximador (único/múltiple)	Múltiple

Tabla 15: Configuración del experimento ISQL4.

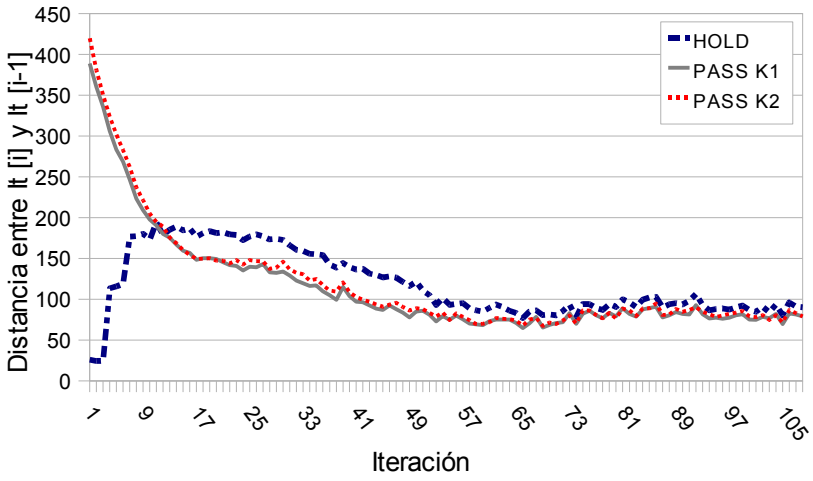

Figura ISQL4-1: Evolución de la distancia entre iteraciones.

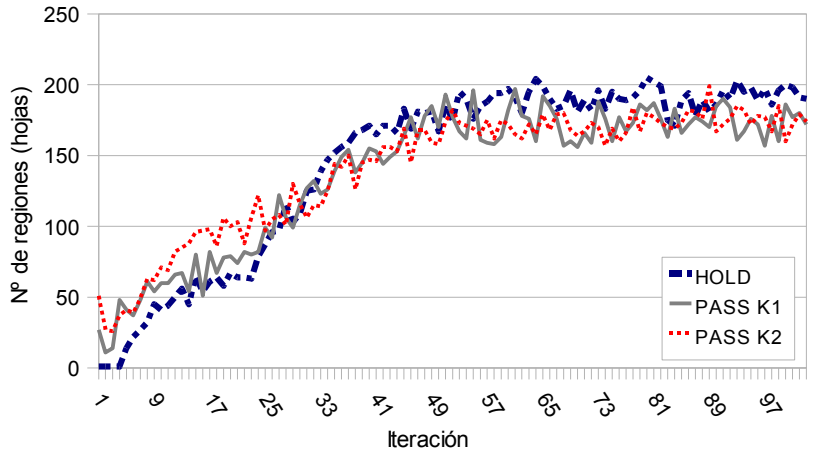

Figura ISQL4-2: Evolución del número de hojas del estimador.

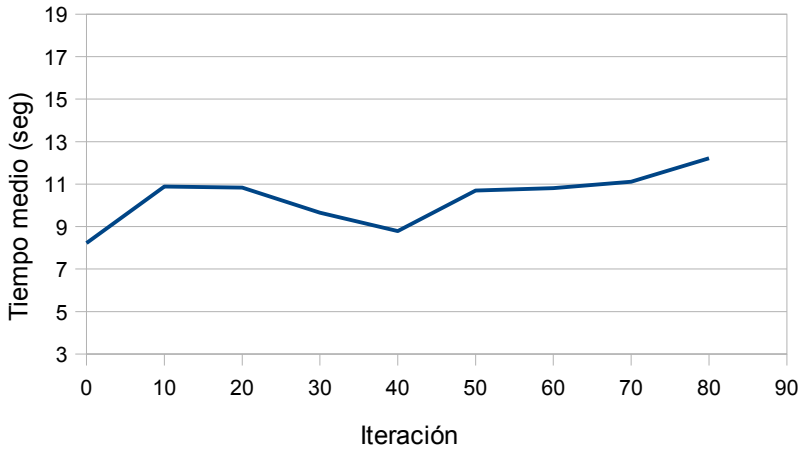

Figura ISQL4-3: Evolución de la duración media por iteración.

Experimento ISQL-5

Este experimento es equivalente al experimento ISQL-2 con la diferencia de que esta vez utilizamos un estimador para cada acción, es decir, un aproximador múltiple. La figura ISQL5-1 representa la evolución de la distancia entre el aproximador de la iteración N y la N-1. Cada una de las 3 curvas se corresponden con la evolución de los estimadores para las 3 acciones: *Hold*, *Pass Keeper 1* y *Pass Keeper 2*. Las distancias observadas en la gráfica ISQL5-1 disminuyen rápidamente. Observamos que para la acción *hold* primero se produce un crecimiento brusco de la distancia hasta la iteración 8 y, a partir de ahí, comienza a disminuir. La figura ISQL5-2 muestra la evolución del número hojas de cada estimador. En la figura ISQL5-3 podemos observar la evolución de la eficacia del estimador obtenido en cada iteración para determinar la política a seguir. Se observa una buena evolución de la política obtenida muy parecida a la conseguida en el experimento ISQL-2. Se obtiene una media máxima de 11,45 segundos en la iteración 20.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	0,25
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	X
Función aproximada (discreta [Nº clases]/continua)	12 clases
Aproximador (único/múltiple)	Múltiple

Tabla 16: Configuración del experimento ISQL5.

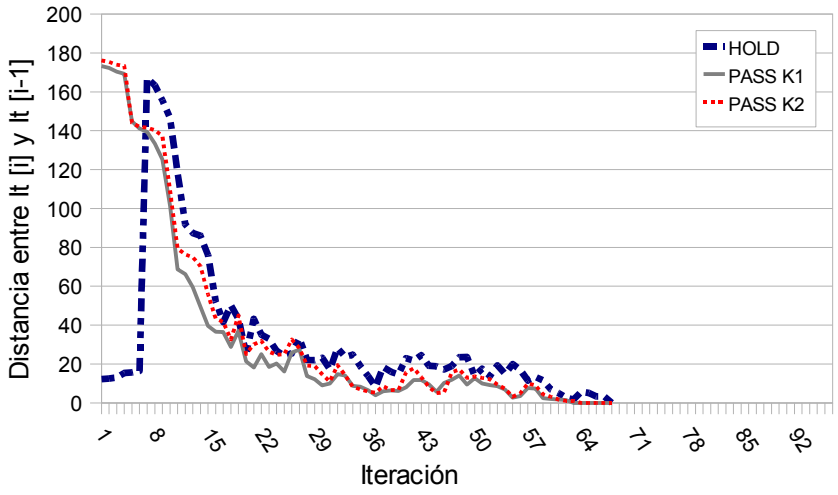

Figura ISQL5-1: Evolución de la distancia entre iteraciones.

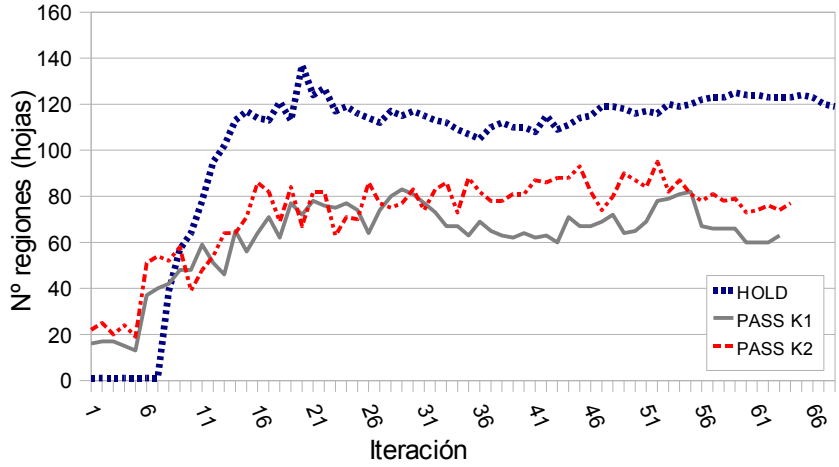

Figura ISQL5-2: Evolución del número de hojas del estimador.

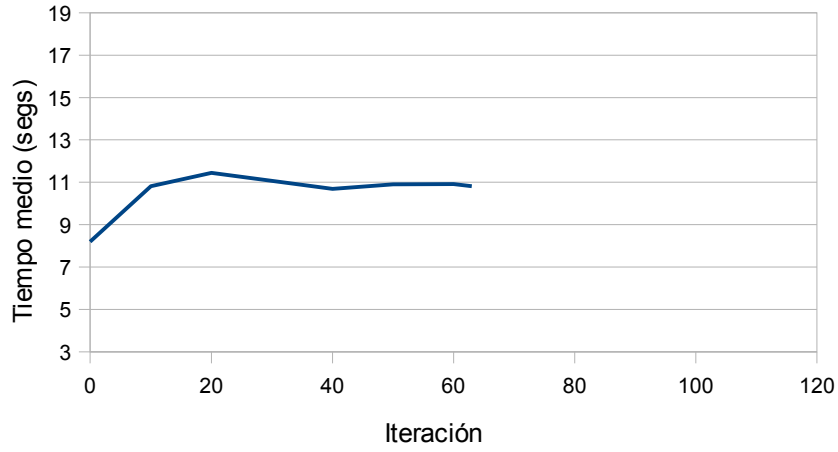

Figura ISQL5-3: Evolución de la duración media por iteración.

4.2.2 Experimentación con M5P

El siguiente apartado reúne los experimentos realizados utilizando M5 como algoritmo para generar los aproximadores de la función Q. La principal ventaja de M5 frente a J48 reside en que, al contrario de J48, M5 admite que el atributo de clase esté representado por valores continuos. Este hecho hace que no sea necesario buscar una discretización del espacio de recompensas que permita la convergencia del 2SRL.

Experimento ISQL-6

La figura ISQL6-1 muestra una disminución logarítmica de la distancia entre iteración, llegando a un mínimo cercano a 0. Por otra parte la figura ISQL6-2 muestra el crecimiento del número de regiones. Podemos observar una evolución más oscilatoria en la evolución del número de regiones que en otros experimentos. Esto puede deberse a que utilizamos un árbol que contiene modelo de regresión en cada hoja y no un árbol de regresión con la media del refuerzo de las instancias de la hoja. En WEKA cuando el parámetro R está activado se genera un árbol de regresión. Cuando R no está activado se genera un árbol de modelos. En este experimento se utiliza un árbol de modelos. La figura ISQL6-3 muestra una evolución muy favorable en la eficacia del estimador generado. La política con mejor media se ha obtenido en la iteración 80 y los episodios que juega tienen una duración media de 17,03 segundos.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	X
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	Modelos
Función aproximada (discreta [Nº clases]/continua)	Continua
Aproximador (único/múltiple)	Único

Tabla 17: Configuración del experimento ISQL6.

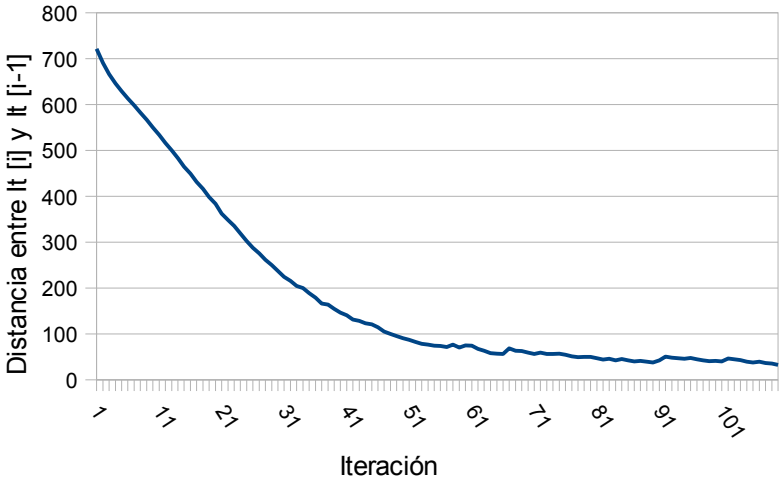

Figura ISQL6-1: Evolución de la distancia entre iteraciones.

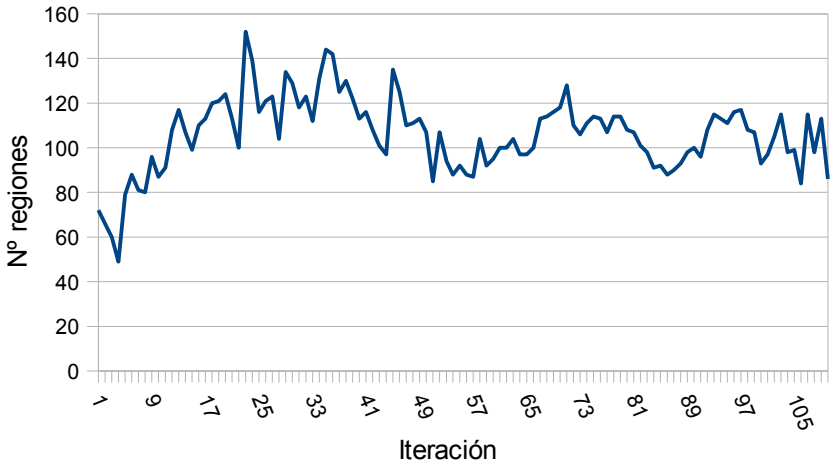

Figura ISQL6-2: Evolución del número de hojas del estimador.

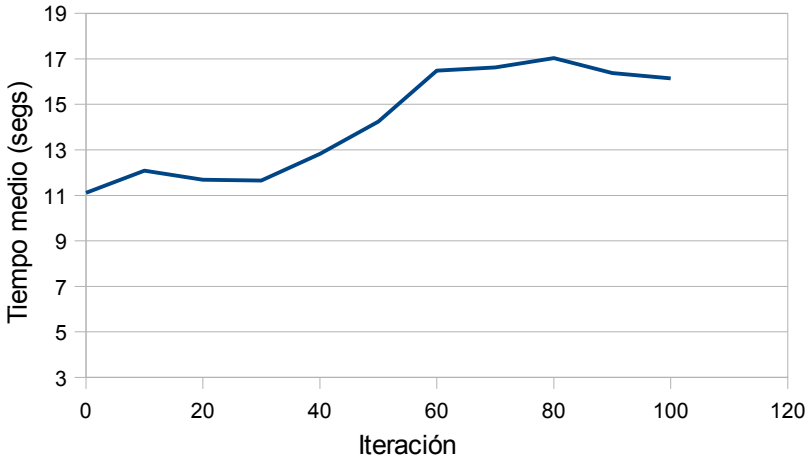

Figura ISQL6-3: Evolución de la duración media por iteración.

Experimento ISQL-7

Este experimento es igual al experimento ISQL-6 pero utilizando un árbol de regresión, el cuál tiene un valor numérico en las hojas. La evolución de la distancia es similar a la del experimento anterior (figura ISQL7-1). Sin embargo, la figura ISQL7-2 muestra una mejor evolución del número de regiones que se estabiliza en torno a 350. La evolución de la efectividad de la política en el simulador es positiva hasta la iteración 50, donde se obtiene una media de 15,03 segundos. Sin embargo, a partir de la iteración 50 la efectividad de la política empieza a empeorar (figura ISQL7-3).

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	X
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	Regresión
Función aproximada (discreta [Nº clases]/continua)	Continua
Aproximador (único/múltiple)	Único

Tabla 18: Configuración del experimento ISQL7.

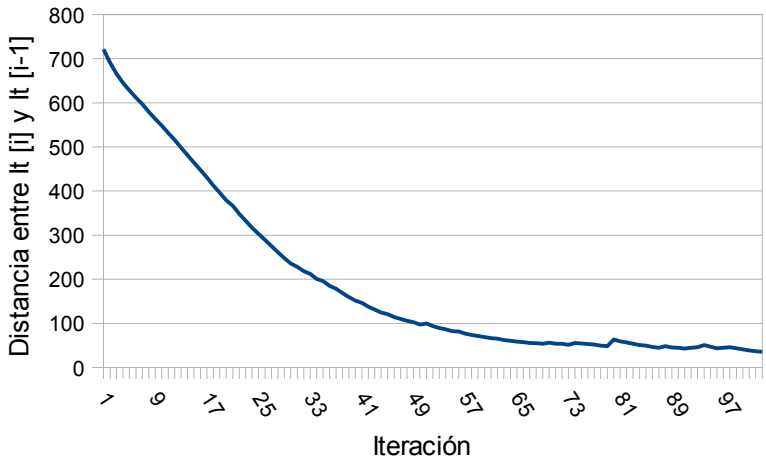

Figura ISQL7-1: Evolución de la distancia entre iteraciones.

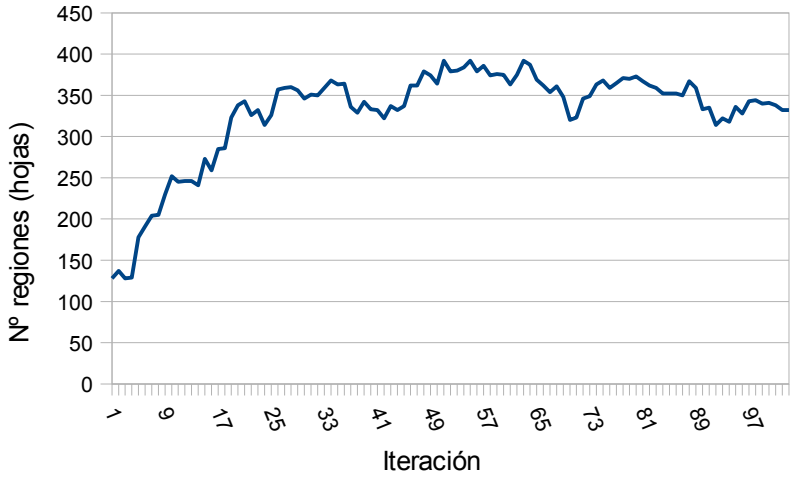

Figura ISQL7-2: Evolución del número de hojas del estimador.

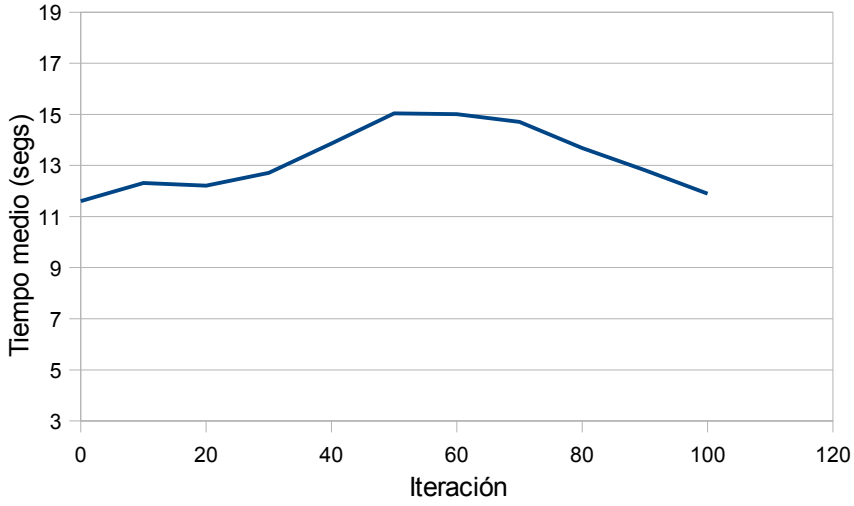

Figura ISQL7-3: Evolución de la duración media por iteración.

Experimento ISQL-8

En este experimento probaremos un número mínimo de hojas menor al utilizado anteriormente. Comprobaremos los efectos que produce sobre los resultados de ISQL y, posteriormente, en MDQL. La figura ISQL8-1 muestra una evolución favorable de la distancia entre iteraciones. En cambio, la figura ISQL8-2 muestra una inestabilidad en la evolución del número de regiones. Para la acción *hold* parece que el número de regiones aumenta, aunque con muchas oscilaciones. Por el contrario, las acciones "*pass*" no evolucionan. Con respecto a la evolución de la efectividad de la política resultante, observamos en la figura ISQL8-3 que la evolución en esta fase si es favorable. El único inconveniente es que la falta de evolución en el número de regiones puede indicar un mal funcionamiento de los aproximadores aplicados en la fase MDQL.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	X
M (Mínimo numero de instancias por hoja)	4
R (regresión/modelos)	Modelos
Función aproximada (discreta [Nº clases]/continua)	Continua
Aproximador (único/múltiple)	Múltiple

Tabla 19: Configuración del experimento ISQL8.

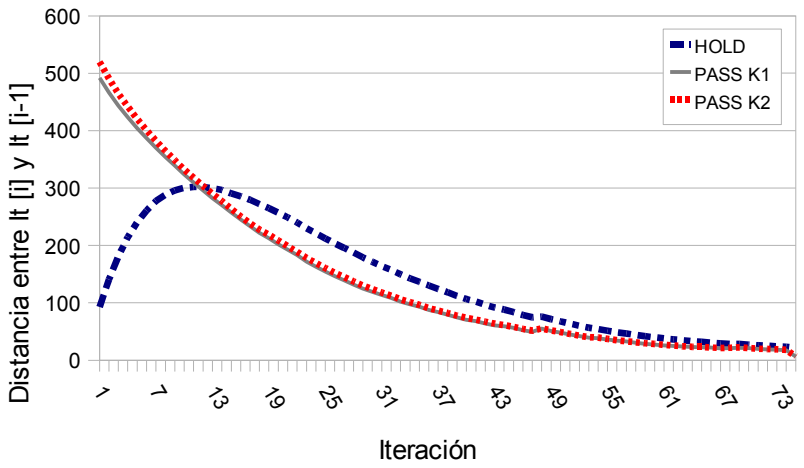

Figura ISQL8-1: Evolución de la distancia entre iteraciones.

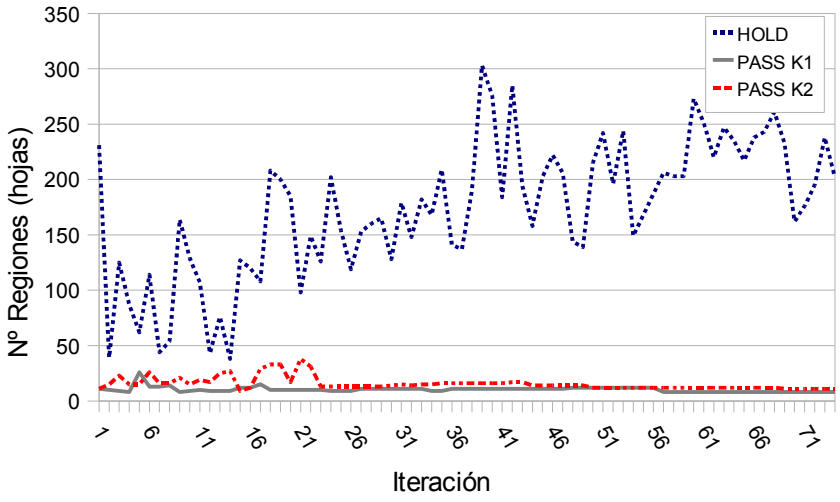

Figura ISQL8-2: Evolución del número de hojas del estimador.

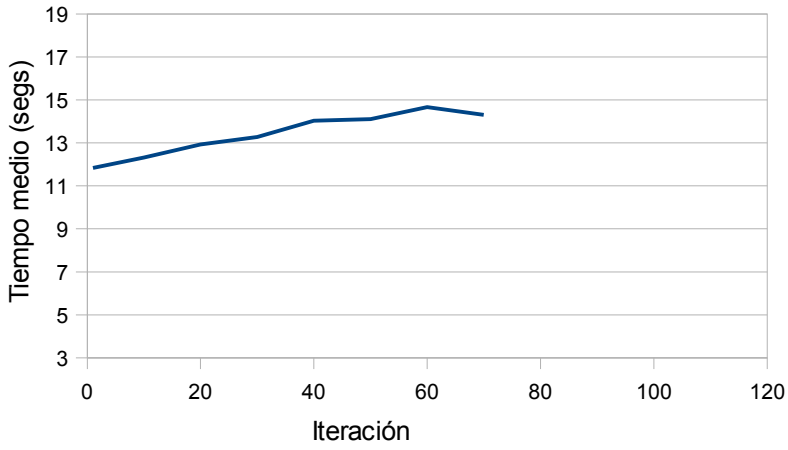

Figura ISQL8-3: Evolución de la duración media por iteración.

Experimento ISQL-9

En este experimento utilizamos la misma configuración utilizada en ISQL-8, a excepción del número mínimo de instancias por hoja, que hemos aumentado a 100. Los resultados son muy parecidos a los obtenidos en ISQL-8, destacando una ligera tendencia de aumento del número de regiones, que en el anterior experimento era nula.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	X
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	Modelos
Función aproximada (discreta [Nº clases]/continua)	Continua
Aproximador (único/múltiple)	Múltiple

Tabla 20: Configuración del experimento ISQL9.

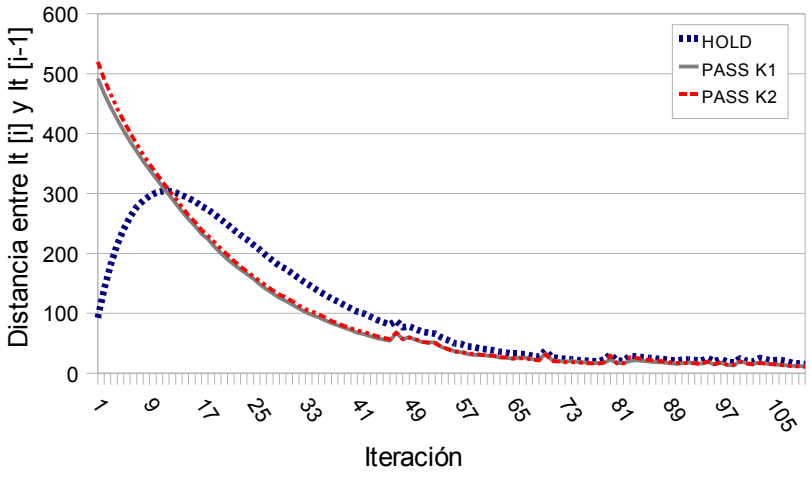

Figura ISQL9-1: Evolución de la distancia entre iteraciones.

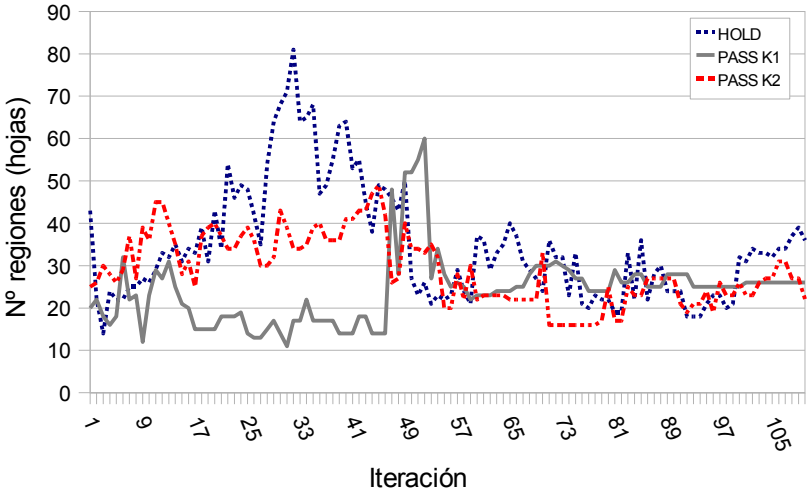

Figura ISQL9-2: Evolución del número de hojas del estimador.

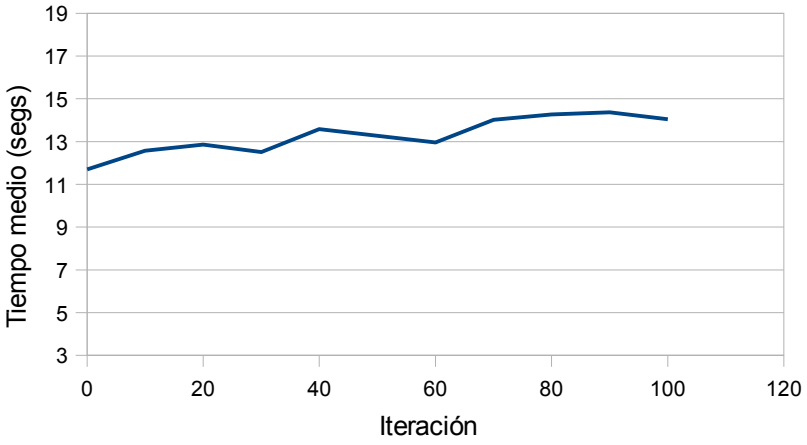

Figura ISQL9-3: Evolución de la duración media por iteración.

Experimento ISQL-10

Con este experimento y el siguiente intentamos mejorar la evolución del número de regiones de los estimadores para que, de esa forma, sean más útiles en la fase MDQL. Teniendo en cuenta los resultados observado en el experimento ISQL-7, tratamos de utilizar árboles de regresión, y no de modelos, para estabilizar la evolución de las regiones definidas por los estimadores. En este caso probamos con un número mínimo de instancias por hoja de 50. Como se puede observar en la figura ISQL10-2, se ha conseguido una tendencia ascendente en la evolución del número de regiones de los estimadores generados en cada iteración. Para la acción *hold* la tendencia es mucho más clara. Para las acciones *pass* el crecimiento existe, pero es más tenue. Con respecto a la evolución de la efectividad de la política la figura muestra una tendencia muy favorable, llegando en la iteración 90 a tener un pico de 15,83 segundos de media.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	X
M (Mínimo numero de instancias por hoja)	50
R (regresión/modelos)	Regresión
Función aproximada (discreta [Nº clases]/continua)	Continua
Aproximador (único/múltiple)	Múltiple

Tabla 21: Configuración del experimento ISQL10.

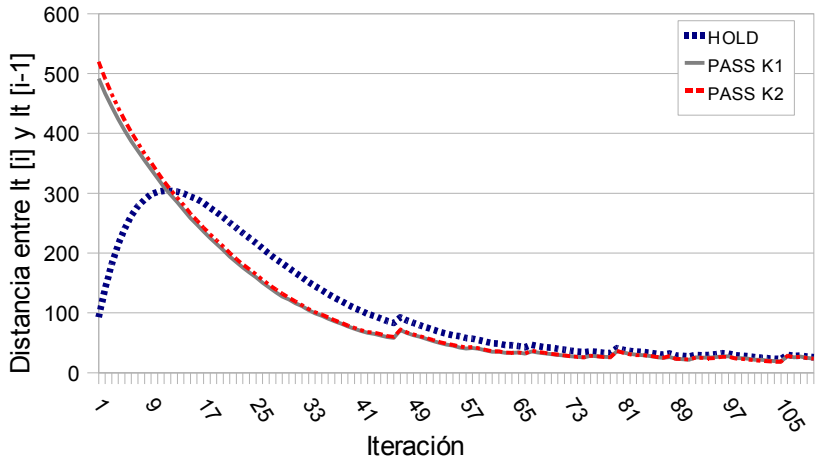

Figura ISQL10-1: Evolución de la distancia entre iteraciones.

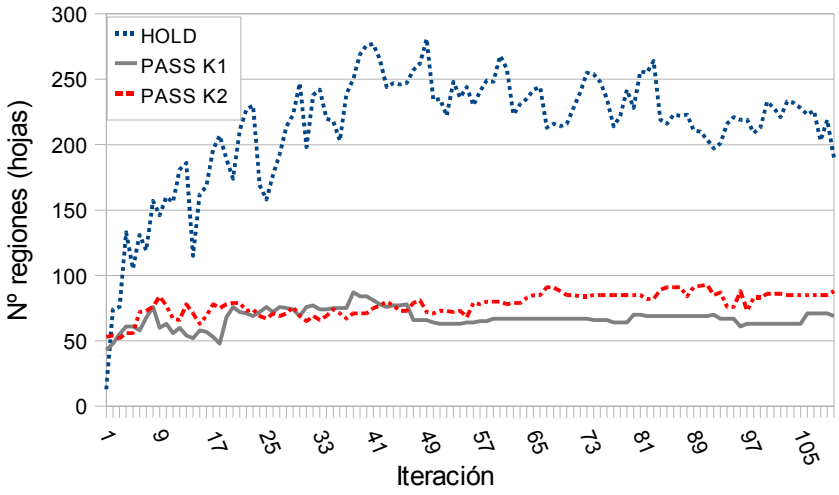

Figura ISQL10-2: Evolución del número de hojas del estimador.

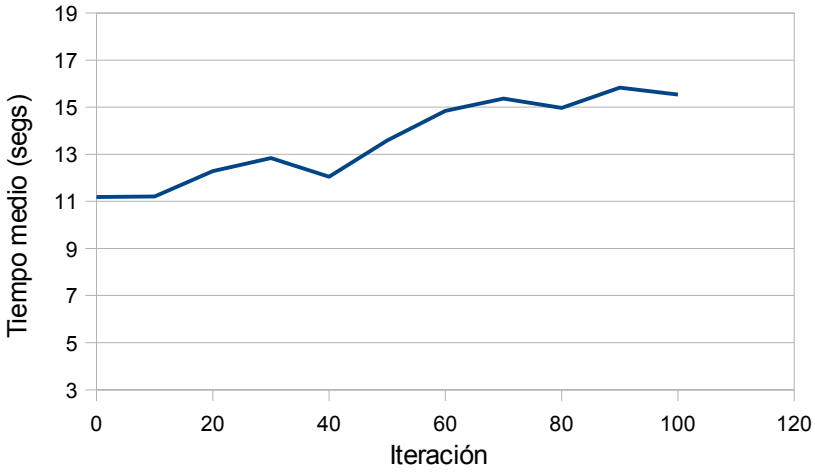

Figura ISQL10-3: Evolución de la duración media por iteración.

Experimento ISQL-11

Este experimento es muy parecido al ISQL-10. Esta vez se ha incrementado el número mínimo de instancias por hoja a 100. Como se puede observar en las gráficas ISQL11-1, ISQL11-2 y ISQL11-3, la evolución es muy parecida a la del experimento ISQL10.

Variables	Valor
α	0,125
γ	1
C (Factor de poda)	X
M (Mínimo numero de instancias por hoja)	100
R (regresión/modelos)	Regresión
Función aproximada (discreta [Nº clases]/continua)	Continua
Aproximador (único/múltiple)	Múltiple

Tabla 22: Configuración del experimento ISQL11.

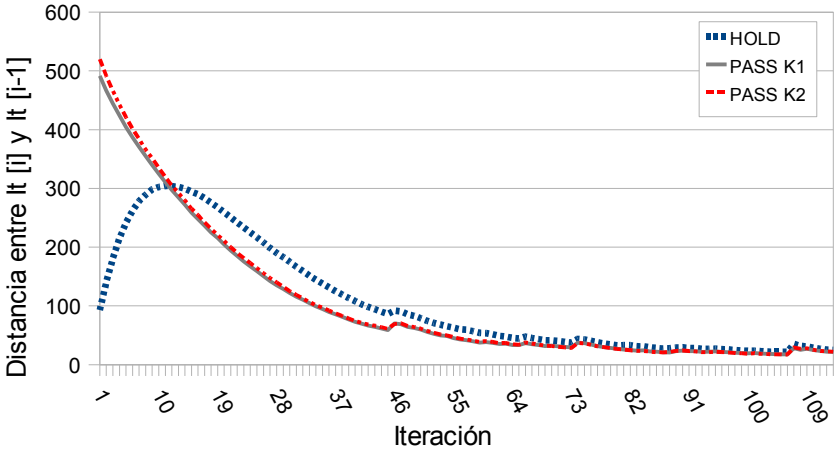

Figura ISQL11-1: Evolución de la distancia entre iteraciones.

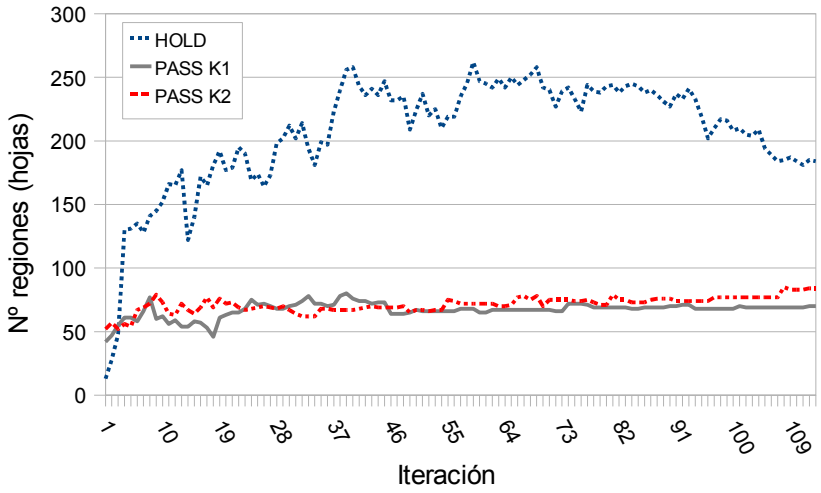

Figura ISQL11-2: Evolución del número de hojas del estimador.

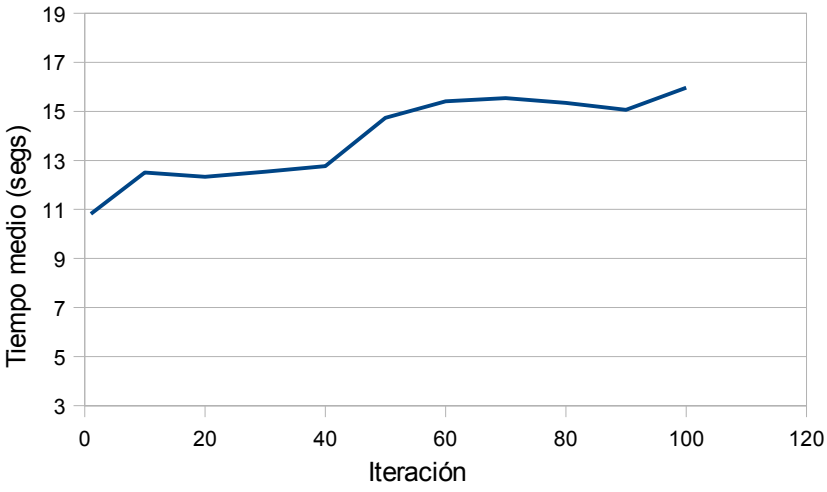

Figura ISQL11-3: Evolución de la duración media por iteración.

4.3 Experimentos realizados en la fase MDQL

En este apartado se comprueba la utilidad de los aproximadores generados en los experimentos de la fase ISQL como discretización del espacio de estados en la fase MDQL. Los algoritmos de aprendizaje por refuerzo que utilizaremos junto con las discretizaciones serán *Q-Learning* y *Q(λ)*.

4.3.1 Experimentos MDQL(J48)-QLearning

A continuación se comentan los resultados obtenidos de aplicar el algoritmo *Q-Learning* sobre las discretizaciones obtenidas a partir del algoritmo ISQL. Con este paso se completa el algoritmo 2SRL, que nos permitirá evaluar la validez de este método en el dominio de la *Keepaway Soccer*. En este bloque se reúnen todos los experimentos que utilizan J48 como aproximación de la función Q.

Experimento MDQL-1

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-1. Los parámetros de Q-Learning utilizados son $\alpha = 0{,}125$, $\gamma = 1$. Estos parámetros se repiten para todos los experimentos MDQL realizados. La estrategia de exploración es $\epsilon - greedy$ con un incremento en ϵ de 0,0001 por cada episodio transcurrido. Se ha realizado experimentación inicializando las casillas de la tabla Q a cero (Q = 0) y inicializando la tabla Q con la política estimada por el aproximador de cada iteración ($Q = Q^{iter}$).

La figura MDQL1-1 muestra la evolución en el aprendizaje de la política aprendida mediante *Q-learning* y las discretizaciones obtenidas en las iteraciones 0, 30 y 80 de ISQL. En la parte izquierda de cada figura se puede observar la evolución en la mejora de la duración de los episodios cuando partimos de una tabla Q inicializada con ceros. En la parte derecha se observa la gráfica de evolución cuando utilizamos los valores de las hojas del aproximador para inicializar la tabla Q. Como se puede observar, no se producen grandes mejoras utilizando una u otra inicialización. Si se observan con detenimiento las gráficas, se puede apreciar una pequeña mejora en las iteraciones 30 y 80 cuando inicializamos Q con los valores de las hojas.

La tabla 23 muestra valores de los experimentos expuestos. La fila $\bar{x}_{ISQL}$ muestra la duración media de los episodios jugados en cada iteración con la política aprendida en la fase ISQL. La fila $\bar{x}_{MDQL-Q=0}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con ceros. La fila $\bar{x}_{MDQL-Q^{iter}}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con la política obtenida en la iteración correspondiente de ISQL. Por último, la última fila muestra el número de regiones en las que el estimador de cada iteración divide el espacio de estados. En la figura MDQL1-4 se puede observar gráficamente estos datos. La gráfica muestra la comparativa entre los resultados de las distintas fases de 2SRL y las distintas inicializaciones de Q. Aquí se observa claramente que el mejor rendimiento se obtiene con MDQL inicializando Q con los valores predichos en la hojas del árbol.

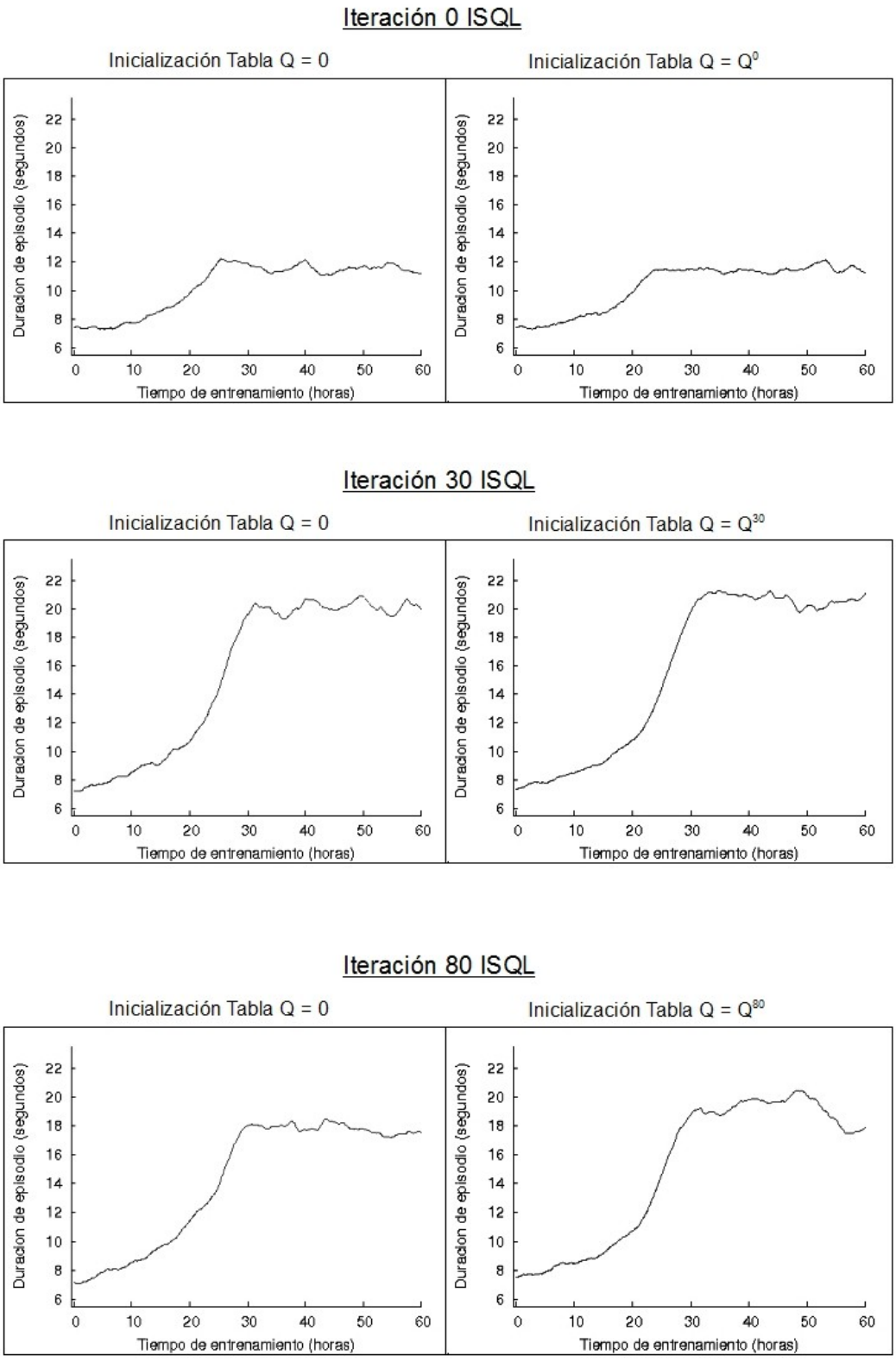

Figura MDQL1-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	30	80
$\bar{x}_{ISQL}$	8,04	10,48	11,63
$\bar{x}_{MDQL-Q=0}$	11,52	20,08	19,05
$\bar{x}_{MDQL-Q^{iter}}$	11,55	20,80	20,29
Nº regiones	79	398	578

Tabla 23: Duración media de episodio utilizando las políticas obtenidas en MDQL1.

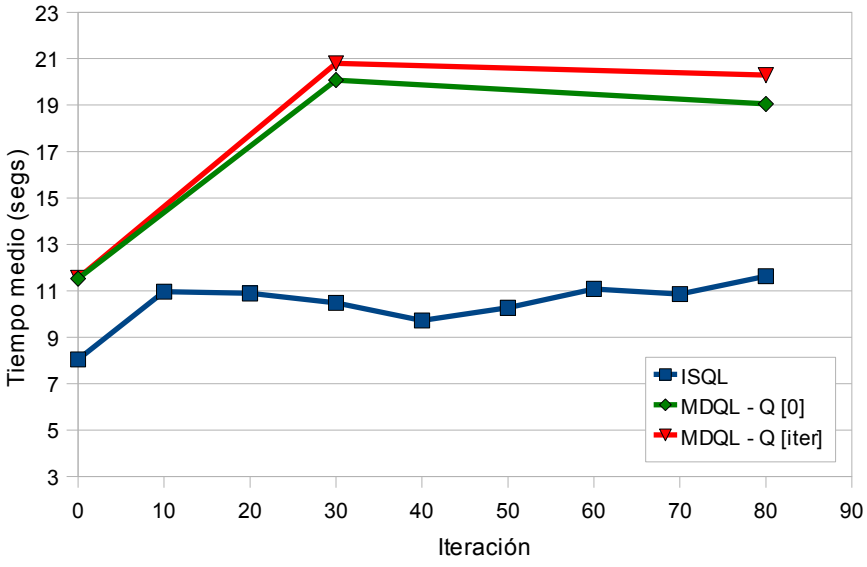

Figura MDQL1-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-2

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-2. Se ha realizado experimentación inicializando la tabla Q con la política estimada por el aproximador de cada iteración ($Q = Q^{iter}$).

Las figura MDQL2-1 muestra la evolución en el aprendizaje de la política aprendida mediante *Q-learning* y las discretizaciones obtenidas en las iteraciones 0, 20 y 50. La tabla 24 muestra con los valores de los experimentos expuestos. La fila $\bar{x}_{ISQL}$ muestra la duración media de los episodios jugados en cada iteración con la política aprendida en la fase ISQL. La fila $\bar{x}_{MDQL-Q^{iter}}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con la política obtenida en la iteración correspondiente de ISQL. Por último, la última fila muestra el número de regiones en las que el estimador de cada iteración divide el espacio de estados. En la figura MDQL2-2 se puede observar gráficamente estos datos. La gráfica muestra la comparativa entre los resultados de las distintas fases de 2SRL. Observamos como 2SRL mejora la política inicial obtenida por ISQL. Con este experimento comprobamos que la discretización del espacio de recompensas que utilicemos junto con J48 es determinante para obtener un buen rendimiento de 2SRL. En el experimento MDQL-1, con una discretización de 88 clases, obtuvimos mejores políticas que en el presente experimento, en donde usamos una discretización del espacio de recompensas con 12 clases.

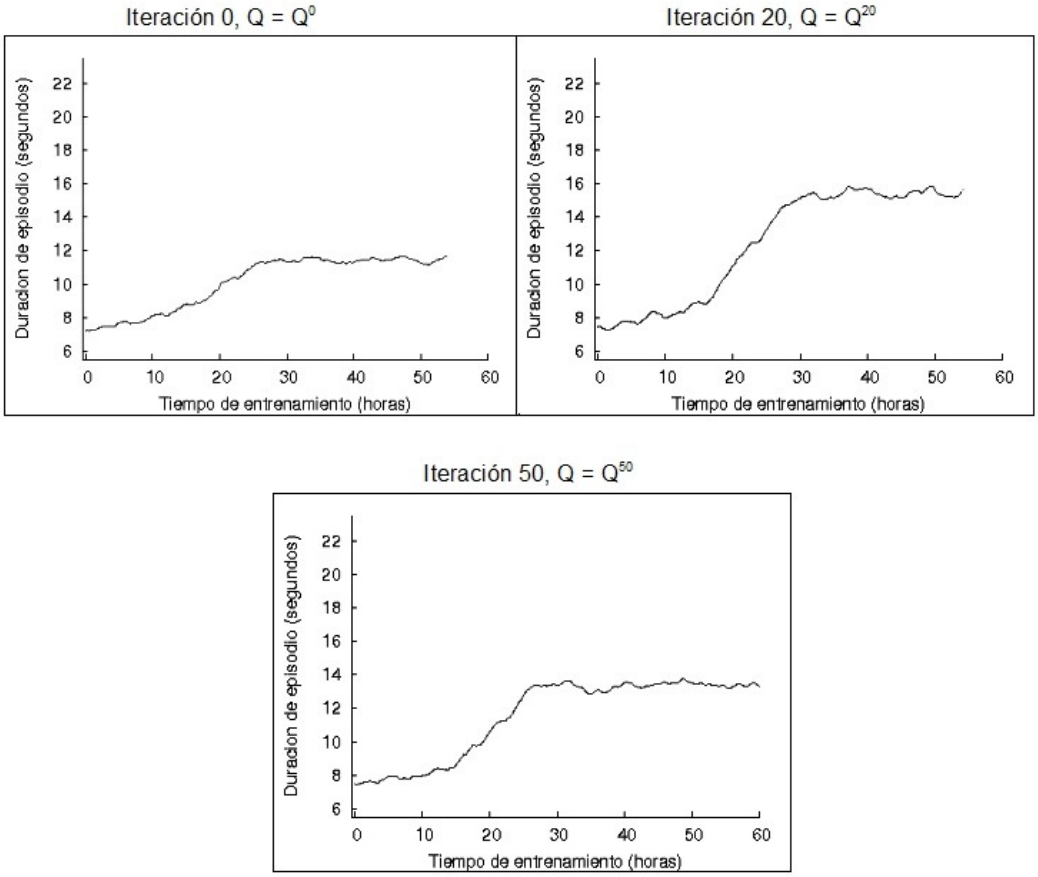

Figura MDQL2-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	20	52
$\bar{x}_{ISQL}$	8,20	11,33	10,43
$\bar{x}_{MDQL-Q^{iter}}$	11,45	15,48	13,33
Nº regiones	39	245	253

Tabla 24: Duración media de episodio utilizando las políticas obtenidas en MDQL2.

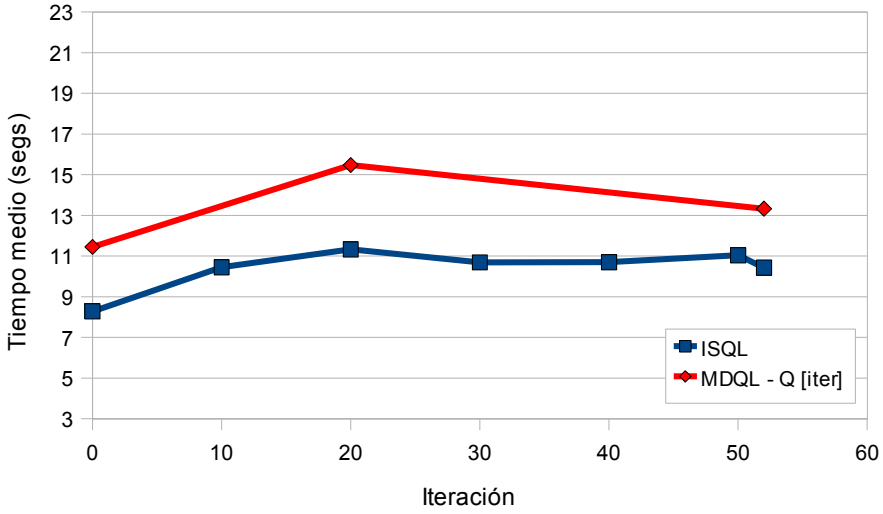

Figura MDQL2-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-3

Este experimento correspondería con la evaluación de las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-3. Dados los resultados obtenidos en la fase ISQL, se ha desestimado la utilidad de las discretizaciones para ser utilizadas junto con *Q-Learning*. De este experimento concluimos que utilizar la función de actualización para dominios deterministas en la fase ISQL para el dominio de la *keepaway* no da buenos resultados. La mala evolución de ISQL se puede observar en la tabla 25 y en su representación gráfica en la figura MDQL3-1.

Iteración	0	20	70	100
$\bar{x}_{ISQL}$	8,09	5,32	4,47	3,97
Nº regiones	49	198	175	182

Tabla 25: Duración media de episodio utilizando las políticas obtenidas en ISQL-3.

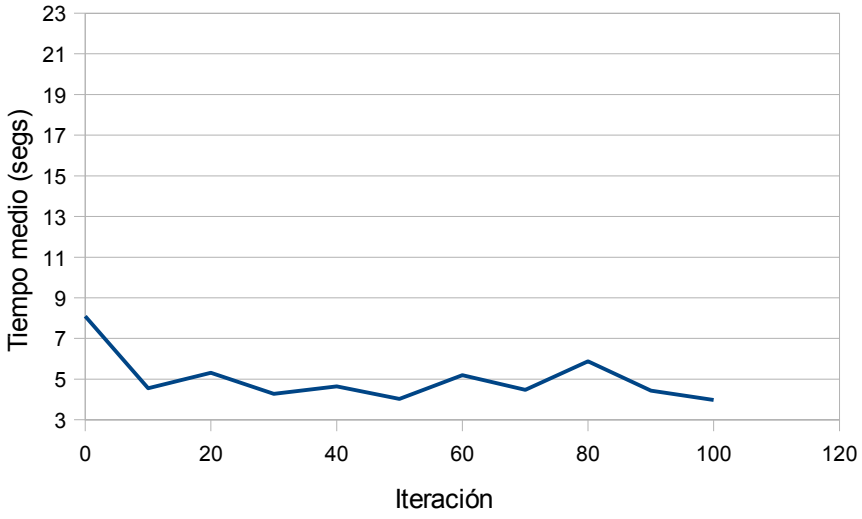

Figura MDQL3-1: Evolución ISQL.

Experimento MDQL-4

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-4. Las figuras MDQL4-1.a y MDQL4-1.b muestran la evolución en el aprendizaje de la política en las iteraciones 0, 10, 60 y 80 respectivamente. En la parte izquierda de las figuras se puede observar la evolución en la mejora de la duración de los episodios cuando partimos de una tabla Q inicializada con ceros. En la parte derecha se observa la gráfica de evolución cuando utilizamos los valores de las hojas del aproximador para inicializar la tabla Q. Observamos que en las iteraciones 10 y 60 no se produce mejora alguna al utilizar la política extraída en ISQL y se obtienen similares resultados inicializando la tabla Q a cero. En cambio, en la iteración 80 observamos una mejora al utilizar la inicialización proporcionada por las hojas del árbol estimador. Como se puede observar, la política de partida en este caso es mejor que en las otras iteraciones, obteniendo una media de 12,22; que al aplicar la fase MDQL crece hasta 17,30 con Q=0 y 18,50 utilizando Q^{iter}.

En la tabla 26 se muestran los valores de los experimentos expuestos en las gráficas anteriores. La fila $\bar{x}_{ISQL}$ muestra la duración media de los episodios jugados en cada iteración con la política aprendida en la fase ISQL. La fila $\bar{x}_{MDQL-Q=0}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con ceros. La fila $\bar{x}_{MDQL-Q^{iter}}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, inicilizando la tabla Q con la política obtenida en la iteración correspondiente de ISQL. Por último, la última fila muestra el número de regiones en las que el estimador de cada iteración divide el espacio de estados. En la figura MDQL4-2 se puede observar gráficamente estos datos. La gráfica muestra la comparativa entre los resultados de las distintas fases de 2SRL y las distintas inicializaciones de Q.

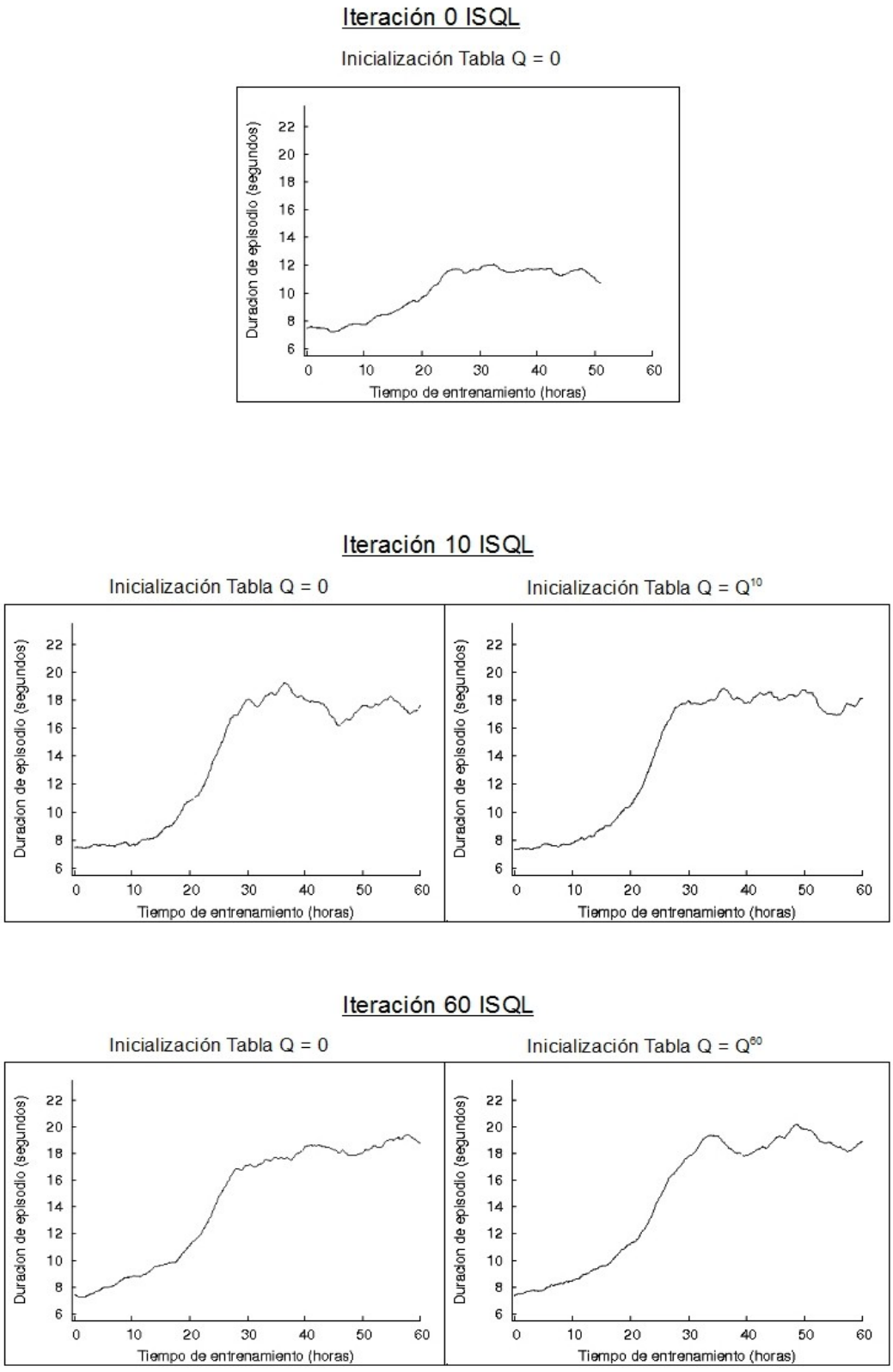

Figura MDQL4-1.a: Gráficas de evolución de aprendizaje con Q-Learning.

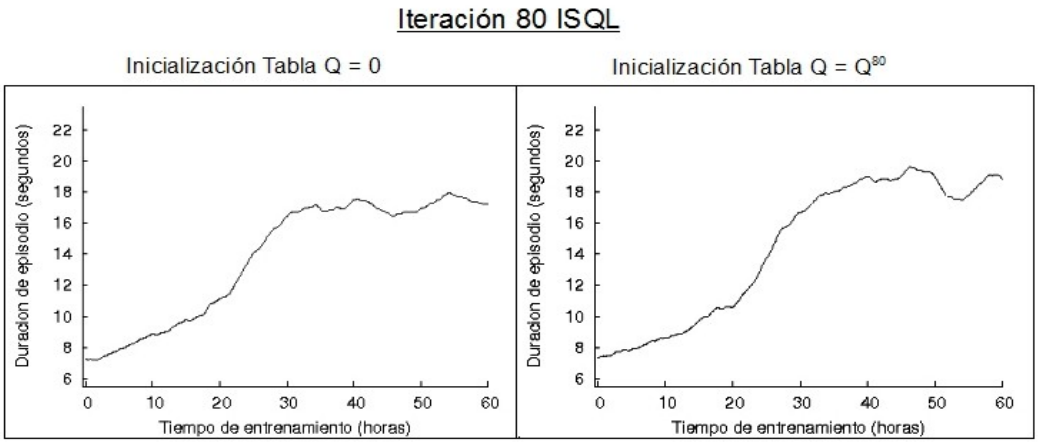

Figura MDQL4-1.b: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	10	60	80
$\bar{x}_{ISQL}$	8,22	10,88	10,81	12,22
$\bar{x}_{MDQL-Q^0}$	11,56	18,10	19,13	17,30
$\bar{x}_{MDQL-Q^{iter}}$	11,58	17,98	19,00	18,50
Nº regiones	79 (1+27+51)*	173 (44+60+69)*	553 (191+197+165)*	566 (202+187+177)*
* Desglose por acción: (hold + pass k1 + pass k2)				

Tabla 26: Duración media de episodio utilizando las políticas obtenidas en MDQL4.

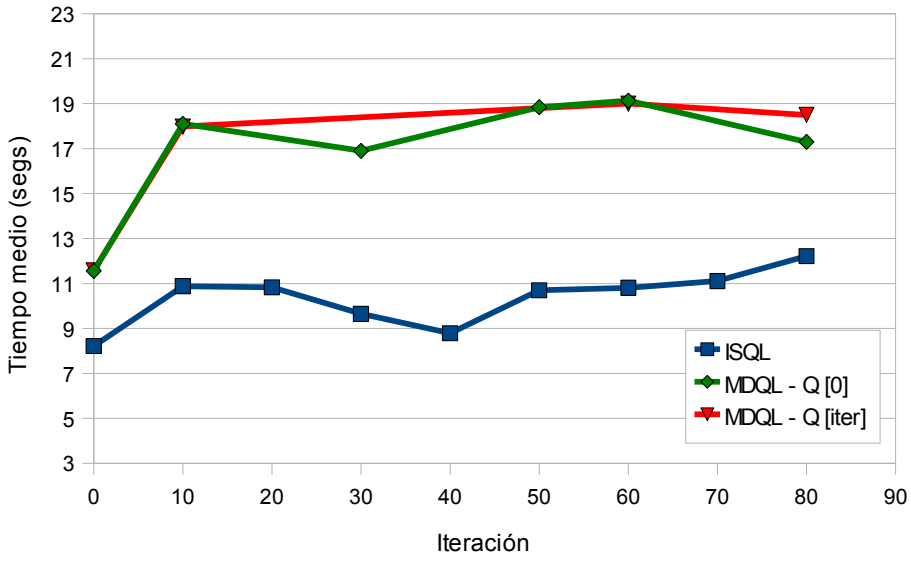

Figura MDQL4-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-5

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-5. Se ha realizado experimentación inicializando las casillas de la tabla Q con la política estimada por el aproximador de cada iteración ($Q = Q^{iter}$). En este experimento comprobamos nuevamente que la discretización con 12 clases funciona peor que la discretización con 88 clases. El experimento MDQL-4 muestra los resultados de la misma configuración utilizada en MDQL-5 con la diferencia de utilizar la discretización del espacio de recompensas con 88 clases. El experimento actual utiliza 12 clases y sus resultados se pueden observar en las figuras MDQL5-1 y MDQL5-2. El algoritmo 2SRL evoluciona positivamente, incrementando la validez de la política en cada iteración, pero los máximos obtenidos están muy por debajo de los obtenidos en el experimento MDQL-4.

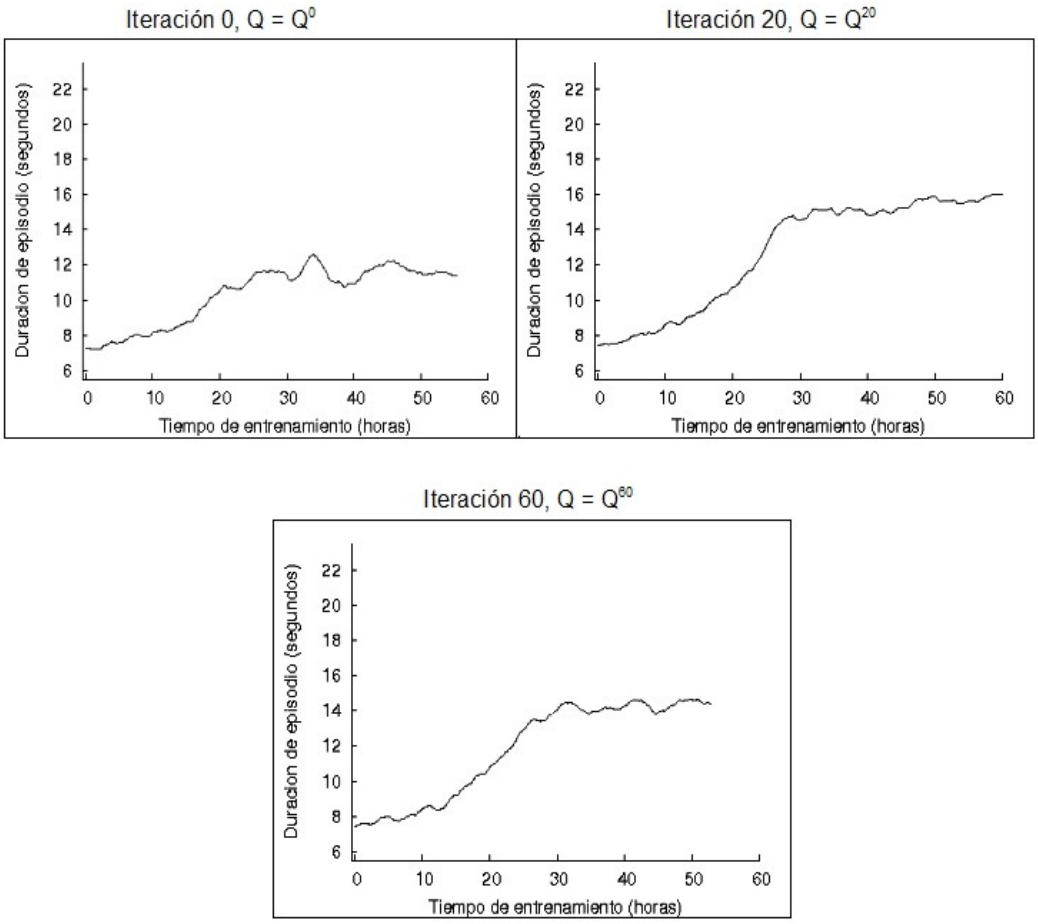

Figura MDQL5-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	20	60
$\bar{x}_{ISQL}$	8,20	11,45	10,92
$\bar{x}_{MDQL-Q^{iter}}$	11,96	15,70	14,28
Nº regiones	39 (1+16+22)*	284 (124+78+82)*	258 (124+60+74)*
* Desglose por acción: (hold + pass k1 + pass k2)			

Tabla 27: Duración media de episodio utilizando las políticas obtenidas en MDQL5.

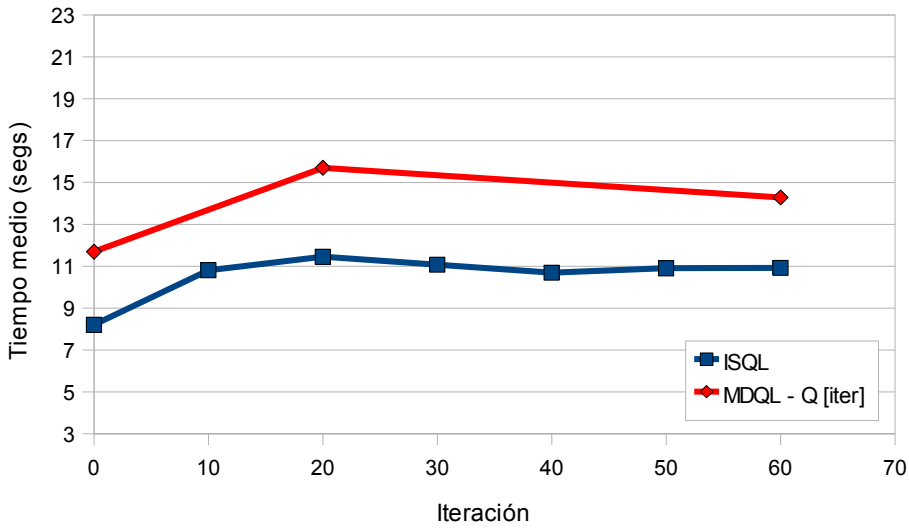

Figura MDQL5-2: Comparativa de evolución entre ISQL y MDQL.

4.3.2 Experimentos MDQL(M5)-QLearning

Al igual que en el bloque anterior se comentan los resultados obtenidos de aplicar el algoritmo *Q-Learning* sobre las discretizaciones obtenidas a partir del algoritmo ISQL. Con este paso se completa el algoritmo 2SRL, que nos permitirá evaluar la validez de este método en el dominio de la *Keepaway Soccer*. En este bloque se reúnen todos los experimentos que utilizan M5 como aproximación de la función Q.

Experimento MDQL-6

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-6. Los parámetros de *Q-Learning* utilizados son $\alpha = 0{,}125$, $\gamma = 1$. La estrategia de exploración es $\epsilon - greedy$ con un incremento en ϵ de 0,0001 por cada episodio transcurrido. Como ya se ha comentado, estos valores de configuración se repiten para todos los experimentos MDQL realizados. En este experimento utilizamos la discretización del espacio de estados proporcionada por árboles de modelos, lo cual significa que no disponemos de un valor en cada nodo hoja del árbol, sino que disponemos de un modelo de regresión lineal que se aplica a todas las instancias clasificadas en esa hoja para predecir el refuerzo o recompensa. Por esa razón no tenemos la posibilidad de utilizar el refuerzo estimado para inicializar cada celda de la tabla Q, disponiendo solo de la posibilidad de inicializar Q a cero. En el correspondiente experimento de ISQL (ISQL-6) se observó la falta de evolución en el número de regiones en la progresión de iteraciones. Como puede observarse en este experimento, esto repercute directamente en la evolución de la fase MDQL, la cual no ha obtenido resultados favorables. Podemos ver como el aplicar M5 con modelos en la hojas ha hecho mejorar la curva de aprendizaje de la política aprendida por el estimador, llegando al mejor resultado obtenido en ISQL con 17,03 segundos de media. Sin embargo las regiones generadas por estos estimadores no son suficientemente buenas para obtener resultados satisfactorios en *Q-Learning*. La tabla 28 y las figuras MDQL6-1 y MDQL6-2 ilustran lo anteriormente descrito.

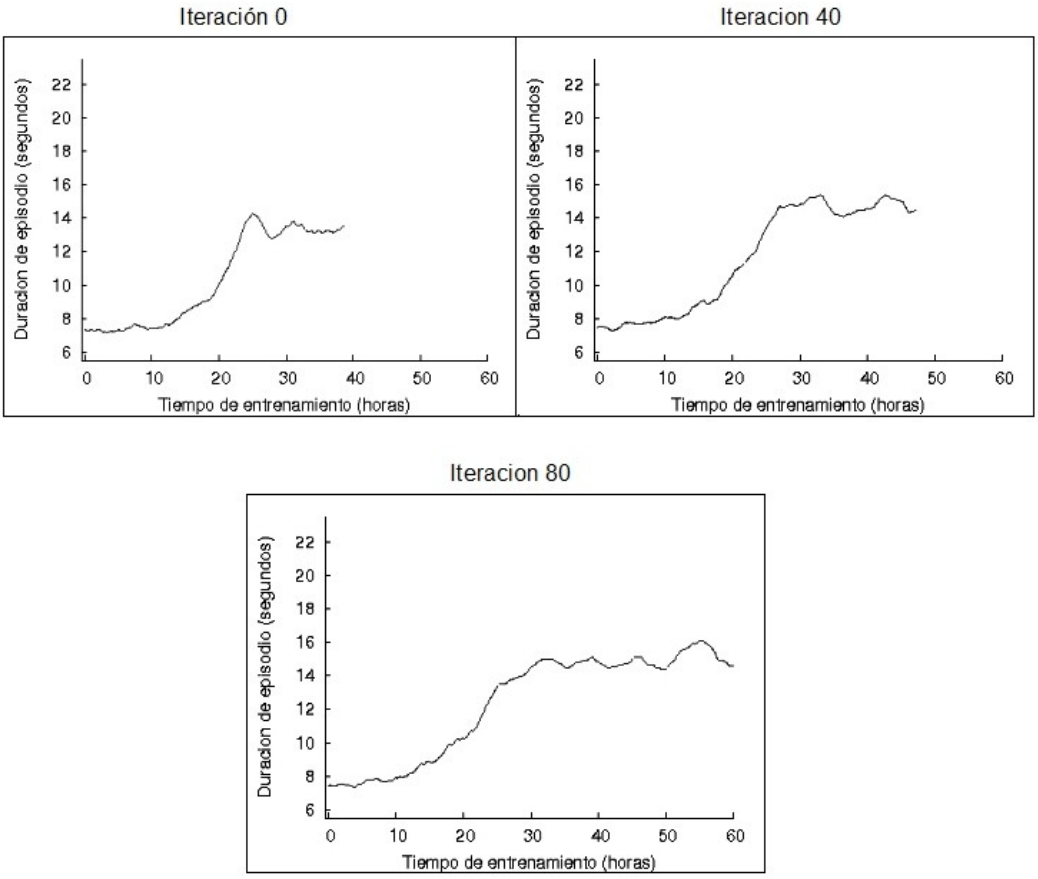

Figura MDQL6-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	40	80
$\bar{x}_{ISQL}$	11,11	12,83	17,03
$\bar{x}_{MDQL}$	13,30	14,80	15,13
Nº regiones	72	116	107

Tabla 28: Duración media de episodio utilizando las políticas obtenidas en MDQL6.

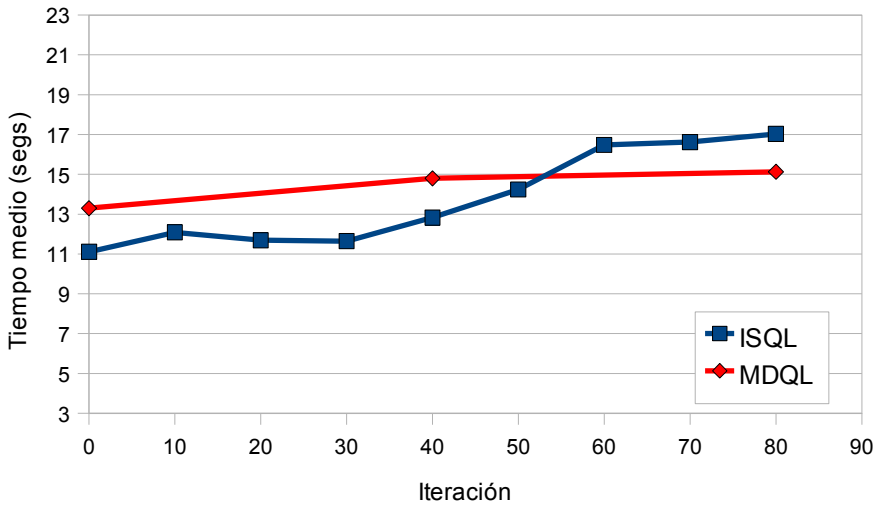

Figura MDQL6-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-7

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-7. En este experimento comprobamos si utilizando un árbol de regresión en vez de modelos conseguimos una mejor evolución de la fase MDQL. En esencia, este experimento repite el experimento MDQL-6 pero con regresión. Los resultados observados en la figura MDQL7-1 y MDQL7-2 confirman que los árboles de regresión dan una discretización más útil para *Q-Learning* que los árboles de modelos. La política obtenida en ISQL ha empeorado un poco, aunque sigue una evolución favorable y la fase MDQL si obtiene resultados que están por encima de ISQL en todas la iteraciones. Aún así, los resultados máximos aún están 2 segundos por debajo de los obtenidos con J48.

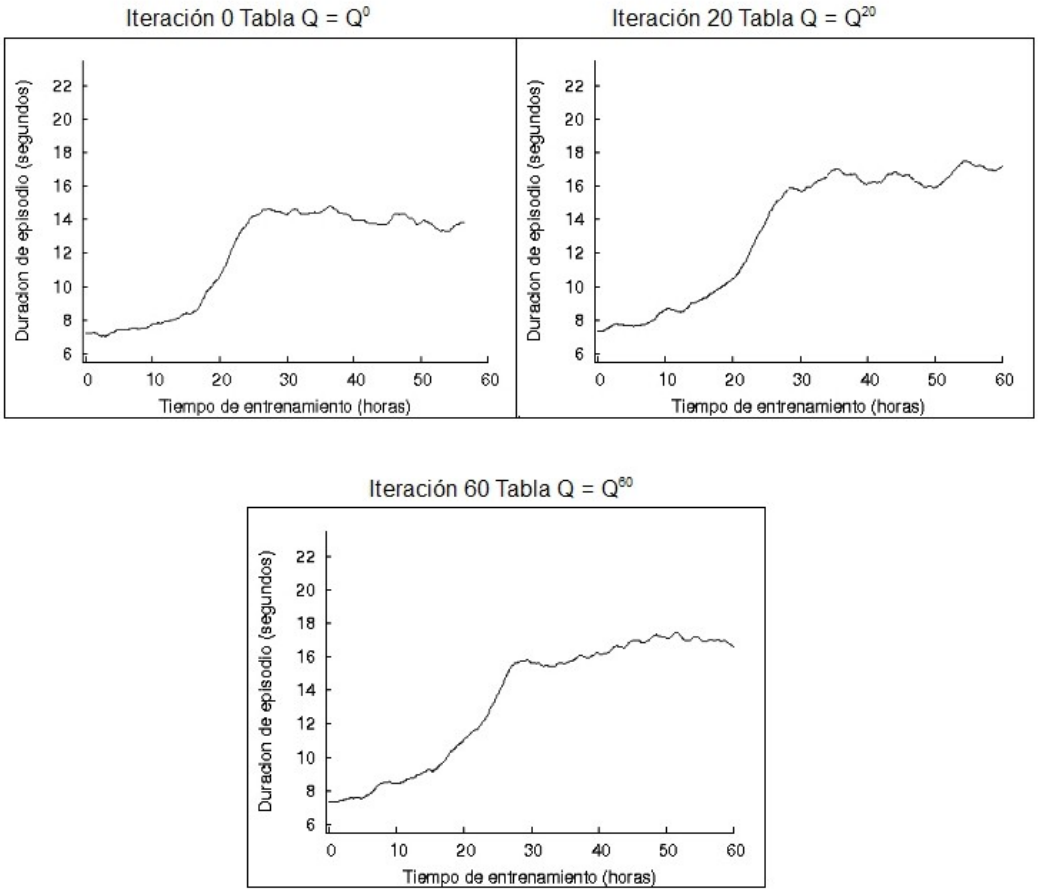

Figura MDQL7-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	20	60
$\bar{x}_{ISQL}$	11,60	12,21	15,01
$\bar{x}_{MDQL}$	13,82	17,11	16,92
Nº regiones	128	343	363

Tabla 29: Duración media de episodio utilizando las políticas obtenidas en MDQL7.

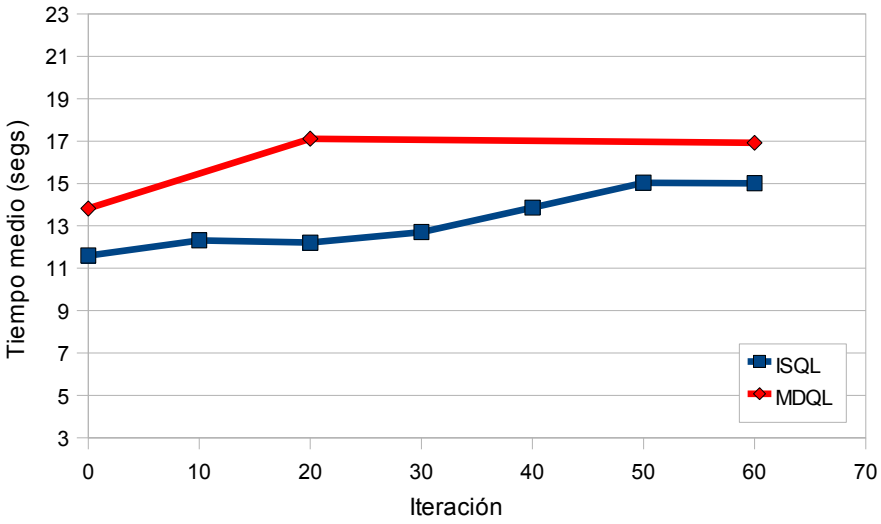

Figura MDQL7-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-8

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-8. Se ha realizado una nueva comprobación para ver de forma más clara la influencia que tiene sobre el resultado utilizar regresión en lugar de modelos y un número mínimo de instancias por hoja más restrictivo.

La tabla 30 muestra los valores de los experimentos realizados. La fila $\bar{x}_{ISQL}$ muestra la duración media de los episodios jugados en cada iteración con la política aprendida en la fase ISQL con los parámetros originales del experimento ISQL-8. La fila $\bar{x}_{MDQL}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, utilizando las discretizaciones directamente generadas en ISQL-8. La fila $\bar{x}_{MDQL-R\,Poda}$ muestra la duración media de los episodios que utilizan una política estable, obtenida en la fase MDQL, usando discretizaciones modificadas a partir de las mismas tuplas que generaron los estimadores de ISQL-8 en cada iteración. Estas modificaciones suponen volver a generar los estimadores, añadiendo la opción de regresión en el algoritmo M5P y dando al número mínimo de instancias por hoja un valor que siempre será igual o mayor a 100. En la figura MDQL8-1 se puede observar gráficamente que los valores obtenidos en $\bar{x}_{MDQL-R\,Poda}$ son mejores que los obtenidos en $\bar{x}_{MDQL}$.

Iteración	0	10	50	60
$\bar{x}_{ISQL}$	11,84	12,32	14,11	14,66
$\bar{x}_{MDQL}$	14,08	13,94	14,42	14,10
$\bar{x}_{MDQL-R\,Poda}$	17,59	18,60	17,16	18,51
Nº regiones	253 (231+11+11)*	136 (107+10+19)*	266 (242+12+12)*	271 (251+8+12)*
* Desglose por acción: (hold + pass k1 + pass k2). Este número de regiones se corresponde con los aproximadores originales, cuyos resultados corresponden a $\bar{x}_{MDQL}$.				

Tabla 30: Duración media de episodio utilizando las políticas obtenidas en MDQL8

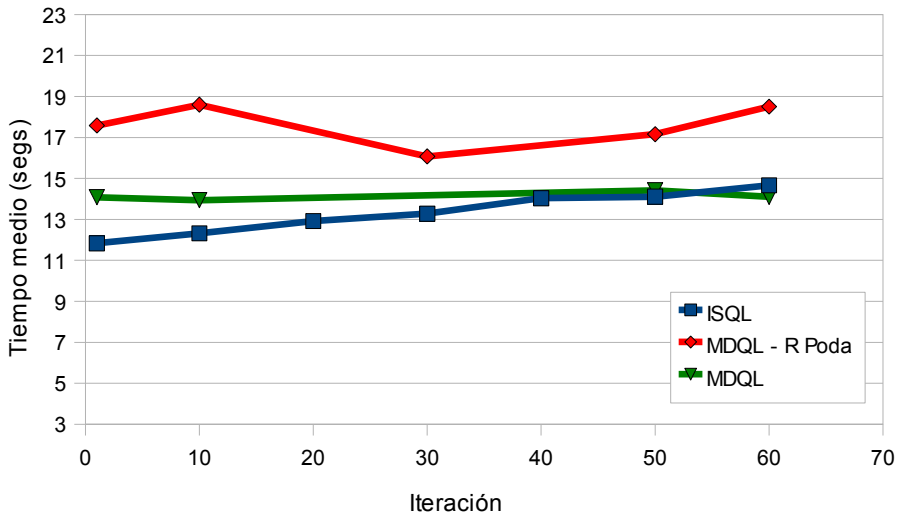

Figura MDQL8-1: Comparativa de evolución entre ISQL, MDQL y MDQL con aproximadores modificados.

Experimento MDQL-9

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-9. En este experimento tenemos otra prueba de que las discretizaciones obtenidas a partir de árboles de modelos no son tan útiles como sería deseable para ser usadas con *Q-Learning*. La figura MDQL9-1, la tabla de valores (tabla 31) y la figura MDQL9-2 demuestran lo comentado anteriormente.

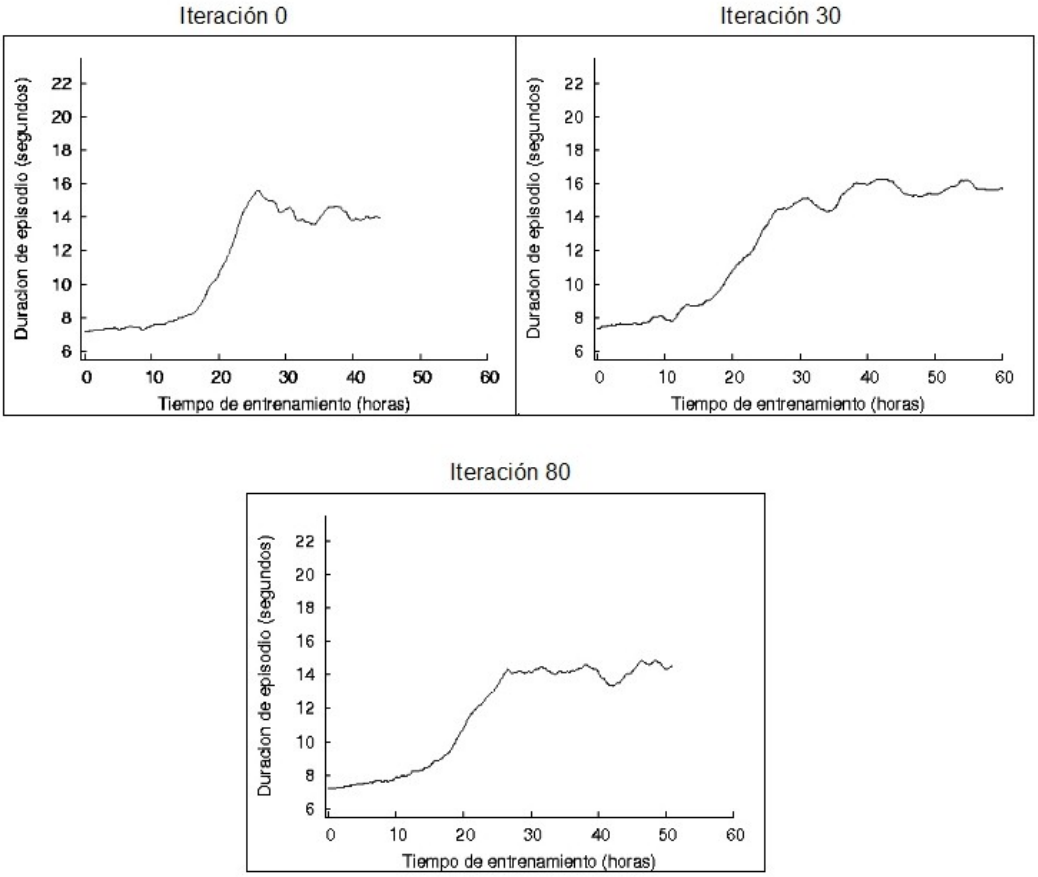

Figura MDQL9-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	30	80
$\bar{x}_{ISQL}$	11,70	12,51	14,27
$\bar{x}_{MDQL}$	13,97	15,58	14,53
Nº regiones	88 (43+20+25)*	121 (71+11+39)*	65 (19+29+17)*
* Desglose por acción: (hold + pass k1 + pass k2)			

Tabla 31: Valor medio de las políticas obtenidas en MDQL9.

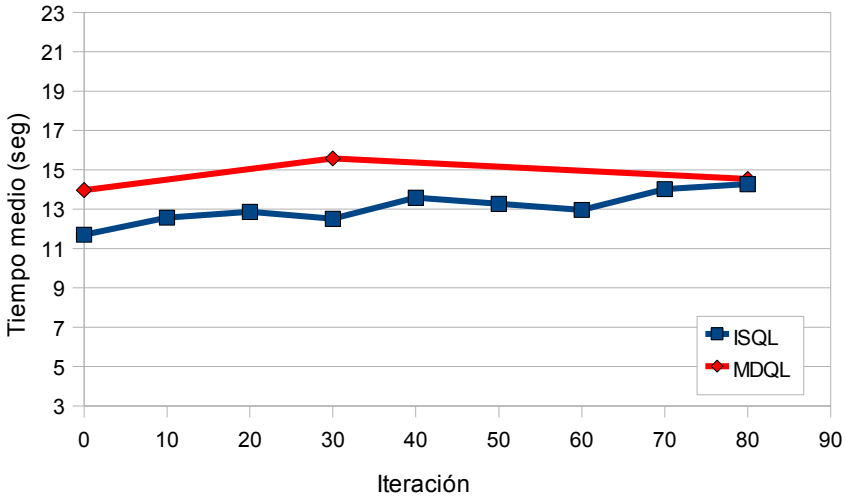

Figura MDQL9-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-10

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQL-10. En este experimento se comprueba como funcionan las discretizaciones de los aproximadores generados con el algoritmo M5 utilizando regresión. Además también se añade un número mínimo de instancias por hoja de 50 instancias por hoja. Este experimento contrasta directamente con el experimento MDQL-9, donde se usaba M5 con modelos. En este caso los resultados mejoran de tal manera que hay una evolución progresiva tanto en la eficacia de la política conseguida como en el número de regiones que genera cada iteración (Figuras MDQL10-1,MDQL10-2 y Tabla 32). También se puede observar los efectos positivos de utilizar la política conseguida en ISQL para inicializar la tabla Q. En la parte izquierda de la figura MDQL10-1 se muestra la gráfica que consigue $\bar{x}_{MDQL-Q^0}$ y en la parte derecha $\bar{x}_{MDQL-Q^{iter}}$.

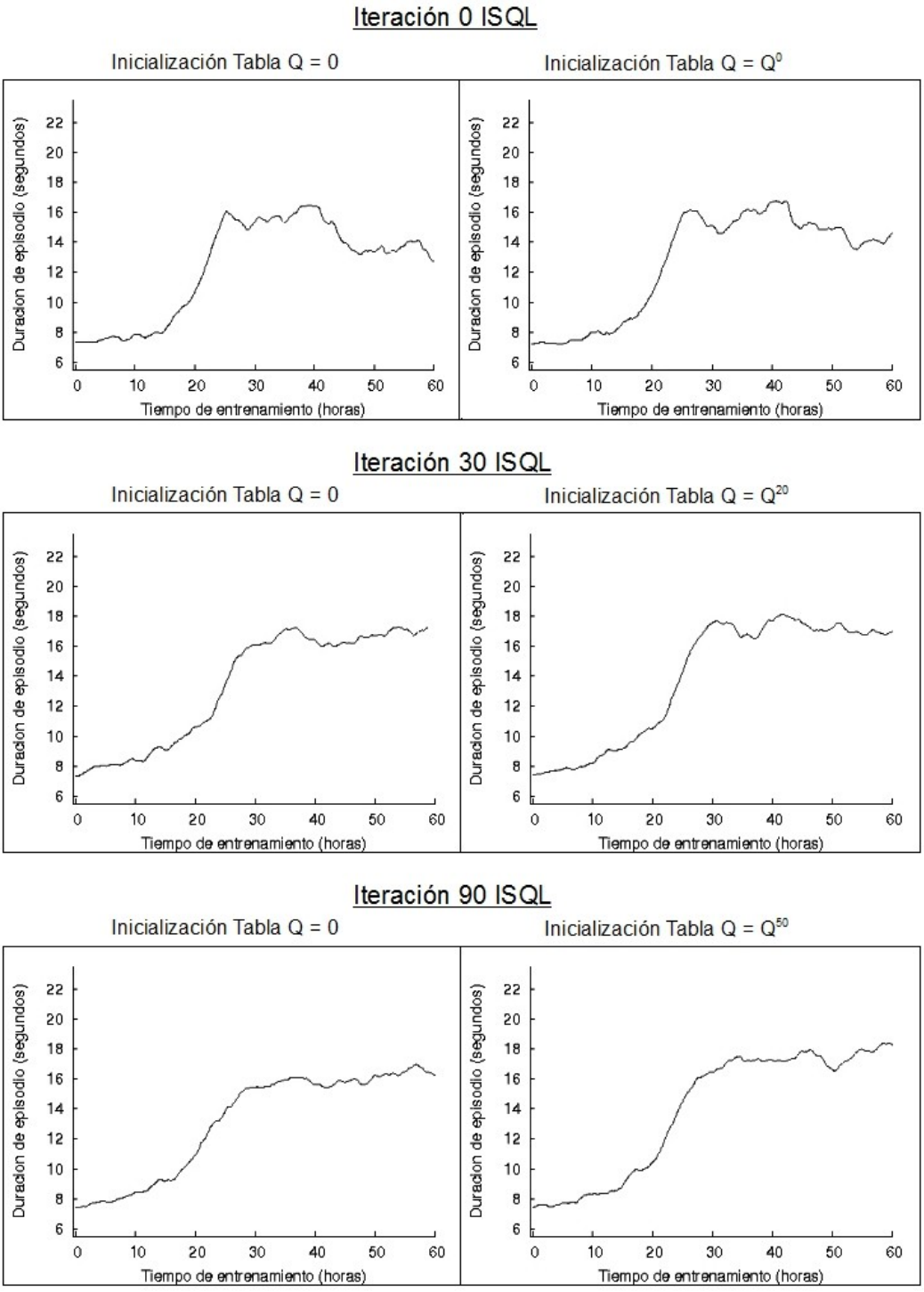

Figura MDQL10-1: Gráficas de evolución de aprendizaje con Q-Learning.

Iteración	0	30	90
$\bar{x}_{ISQL}$	11,18	12,84	15,83
$\bar{x}_{MDQL-Q^0}$	14,04	16,68	16,42
$\bar{x}_{MDQL-Q^{iter}}$	14,24	17,11	17,82
Nº regiones	109 (13+43+53)*	339 (198+76+65)*	366 (204+69+93)*
* Desglose por acción: (hold + pass k1 + pass k2)			

Tabla 32: Valor medio de las políticas obtenidas en MDQL10.

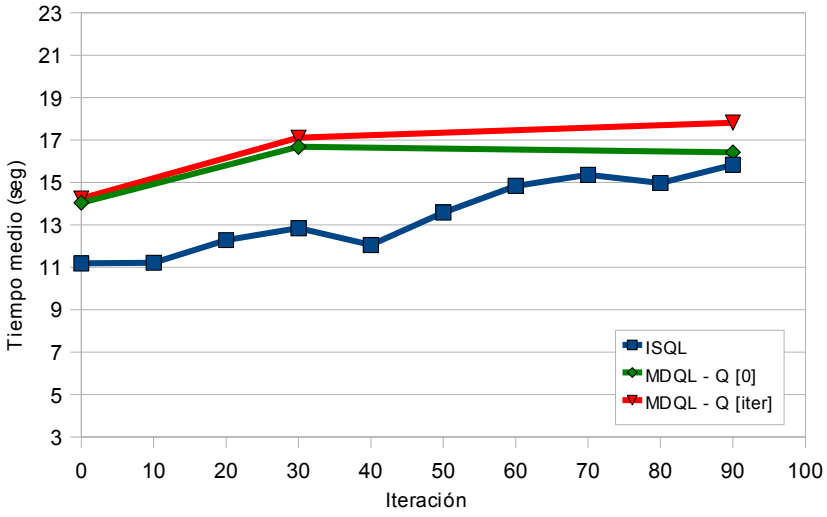

Figura MDQL10-2: Comparativa de evolución entre ISQL y MDQL.

Experimento MDQL-11

En este experimento evaluamos las discretizaciones obtenidas a partir de los estimadores generados en el experimento ISQ-L11. En este experimento se comprueba como funcionan las discretizaciones de los aproximadores generados con el algoritmo M5 utilizando regresión, de la misma manera que el experimento MDQL-10. La diferencia reside en que se añade un número mínimo de instancias por hoja más restrictivo (100 instancias por hoja). En este caso los resultados mejoran sustancialmente en comparación con MDQL-9 y MDQL-10 (Figuras MDQL11-1,MDQL11-2, MDQL11-3 y Tabla 33). Estos resultados son los que más se acercan a los conseguidos con J48. También se puede observar los efectos positivos de utilizar la política conseguida en ISQL para inicializar la tabla Q. En la parte izquierda de las figuras MDQL11-1 y 2 se muestra la gráfica que consigue $\overline{x}_{MDQL-Q^0}$ y en la parte derecha $\overline{x}_{MDQL-Q^{iter}}$.

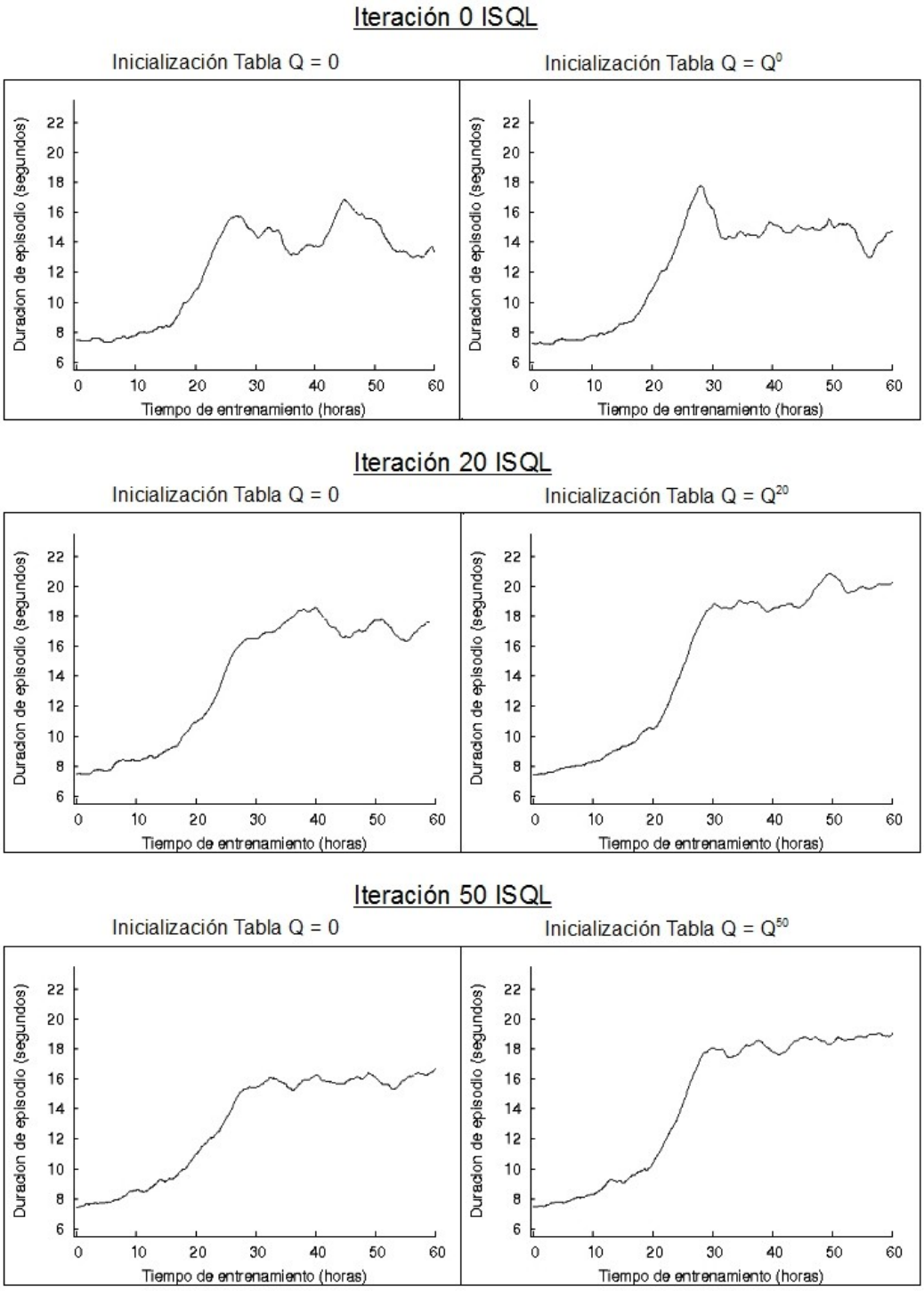

Figura MDQL11-1: Gráficas de evolución de aprendizaje con Q-Learning [iteraciones 0-50].

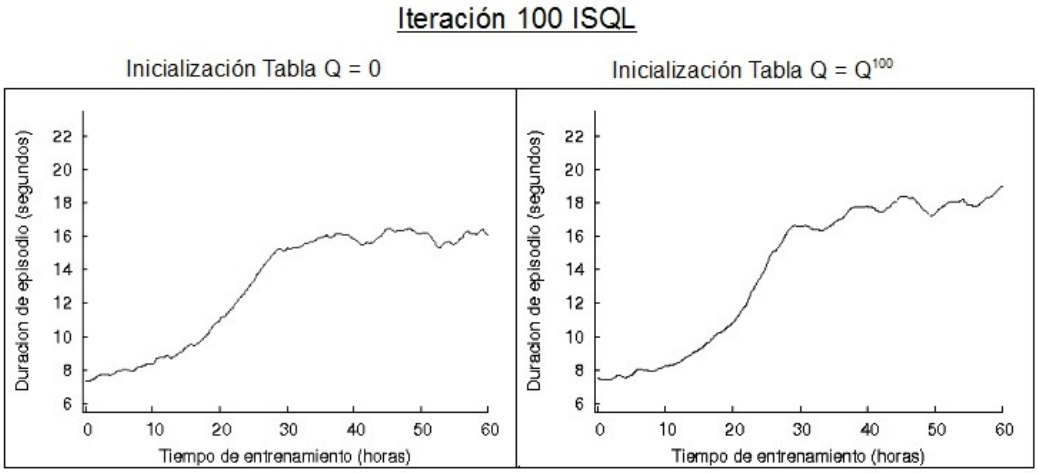

Figura MDQL11-2: Gráficas de evolución de aprendizaje con Q-Learning [iteración 100].

Iteración	0	20	50	100
$\bar{x}_{ISQL}$	10,82	12,33	14,74	15,97
$\bar{x}_{MDQL-Q^0}$	13,29	17,11	16,66	16,07
$\bar{x}_{MDQL-Q^{iter}}$	14,40	19,7	19,03	18,57
Nº regiones	107 (13+42+52)*	317 (179+65+73)*	370 (237+66+67)*	357 (210+70+77)*
* Desglose por acción: (hold + pass k1 + pass k2)				

Tabla 33: Valor medio de las políticas obtenidas en MDQL11.

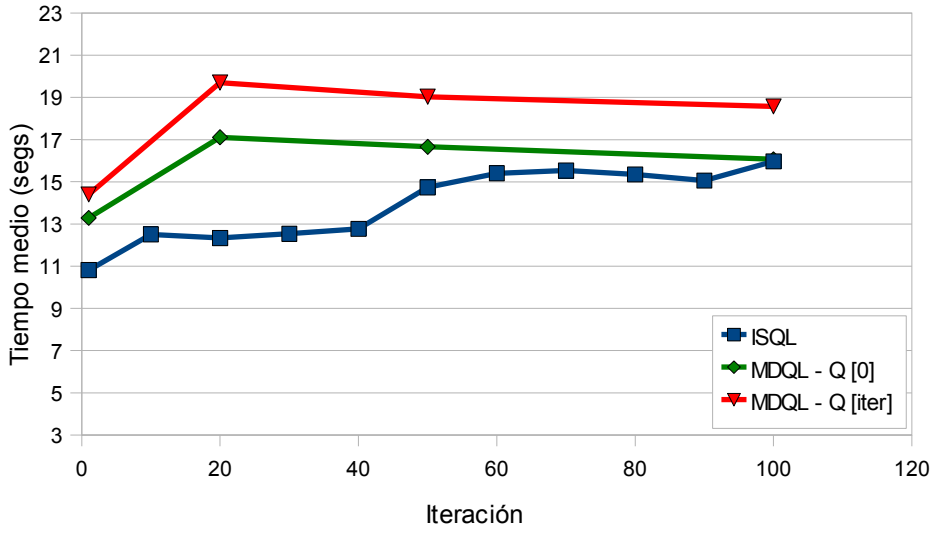

Figura MDQL11-3: Comparativa de evolución entre ISQL y MDQL.

4.3.3 Experimentos MDQL-Q(λ)

A continuación se comprueba el rendimiento del algoritmo Q(λ) junto con la discretización que mejores resultados ha dado con *Q-Learning*. Utilizar *Q-Learning* equivale a utilizar Q(λ) con un valor de $\lambda = 0$. Según lo anterior podemos decir que los anteriores experimentos utilizaron como algoritmo de aprendizaje en la fase MDQL Q(0). En esta sección se comprobará que diferencias pueden surgir cuando aplicamos la técnicas de trazas de elegibilidad para generar métodos intermedios entre *Q-Learning* y *Montecarlo*. Las discretizaciones que utilizaremos para la comparativa Q(0) vs Q(λ > 0) serán las de los experimentos ISQL-1 iteración 30, ISQL-4 iteración 60 y ISQL-11 iteración 20. El valor de lamda que utilizaremos será $\lambda = 0,9$.

En las figuras Lambda-1, Lambda-2 y Lambda-3 observamos un resultado similar. El utilizar trazas de elegibilidad ha provocado en los tres caso un estancamiento prematuro de la curva de aprendizaje. No existen grandes diferencias pero se aprecia en las tres gráficas que las curvas generadas con Q(0.9) están ligeramente por debajo de las curvas generadas por Q(0). Con estos tres ejemplos podemos apreciar como *Q-Learning* funciona mejor sin trazas de elegibilidad en este dominio.

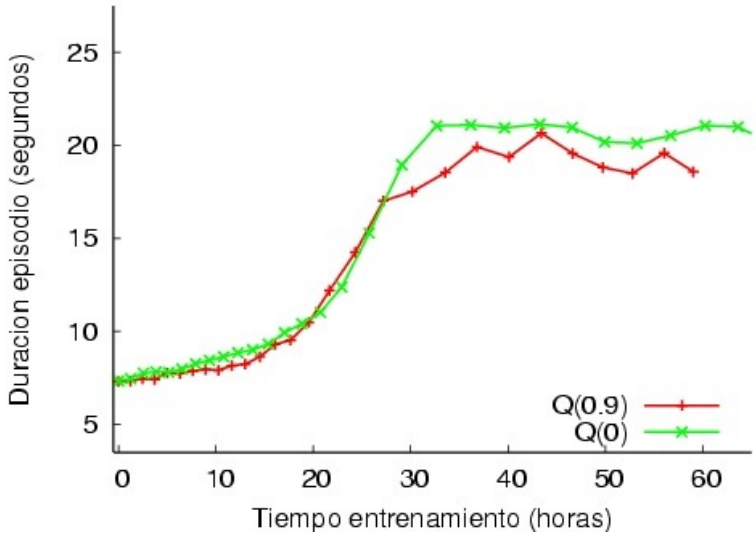

Figura Lambda-1: Comparativa iteración 30 MDQL 1 Q(0) vs Q(λ).

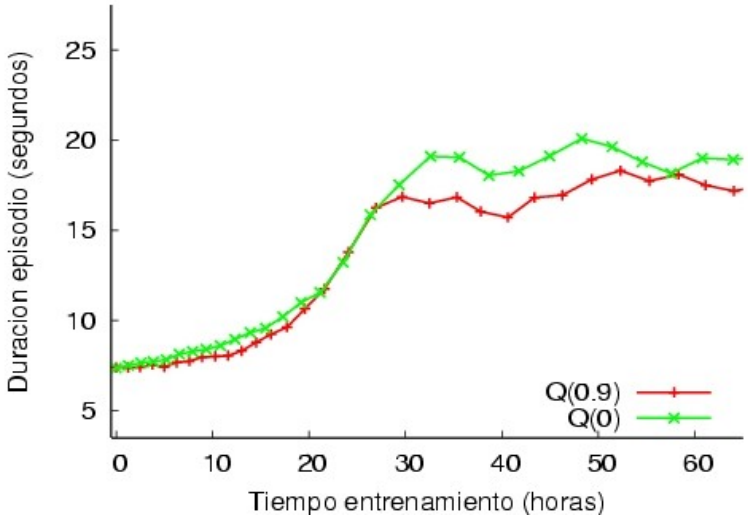

Figura Lambda-2: Comparativa iteración 60 MDQL 4 Q(0) vs Q(λ).

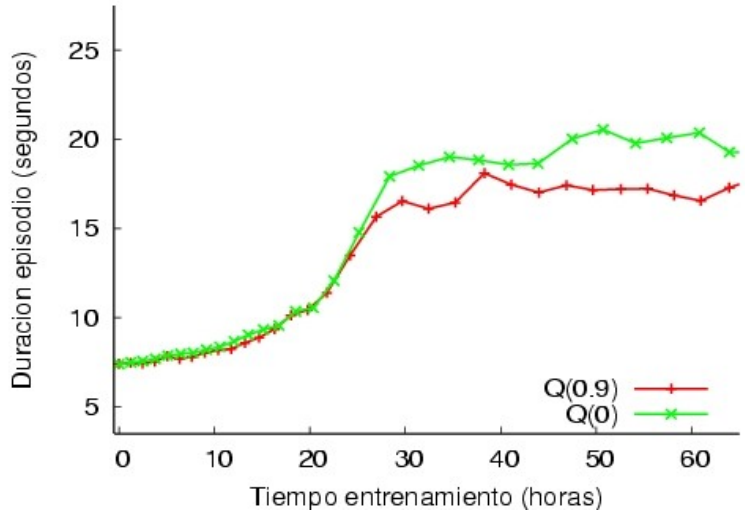

Figura Lambda-3: Comparativa iteración 20 MDQL 11 Q(0) vs Q(λ).

4.4 Comparativa con trabajos anteriores

Este apartado realiza una comparativa entre los resultados obtenidos en el presente proyecto utilizando el algoritmo *Two Steps Reinforcement Learning (2SRL)* y los resultados registrados en el trabajo [García et al, 2007] utilizando *CMAC* y *VQQL*. También se comparan los resultados obtenidos cuando utilizamos como estimador WEKA-M5P (2SRL-M5) y WEKA-J48 (2SRL-J48) con 1 solo estimador para todas las acciones (aproximador único) y 1 estimador para cada acción (3 acciones, 3 estimadores. Aproximador múltiple).

En la figura C-1 podemos observar las curvas de aprendizaje del mejor resultado obtenido con J48 utilizando un solo aproximador y J48 utilizando un aproximador por acción. Como se puede ver, el resultado con un solo aproximador ha conseguido elevar la curva de aprendizaje por encima del resultado con 3 aproximadores. En cambio en la figura C-2 ocurre lo contrario. En los experimentos con M5, el resultado con 3 aproximadores ha sido mejor que el obtenido con solo uno.

En la figura C-3 podemos ver la comparativa entre los mejores resultados de 2SRL-J48, 2SRL-M5, VQQL y CMAC. Los resultados obtenidos son muy satisfactorios, ya que se ha conseguido superar el resultado obtenido por VQQL tanto en 2SRL-J48 como en 2SRL-M5. Sin embargo no se han superado los resultados obtenidos por CMAC, aunque se han obtenido resultados muy cercanos.

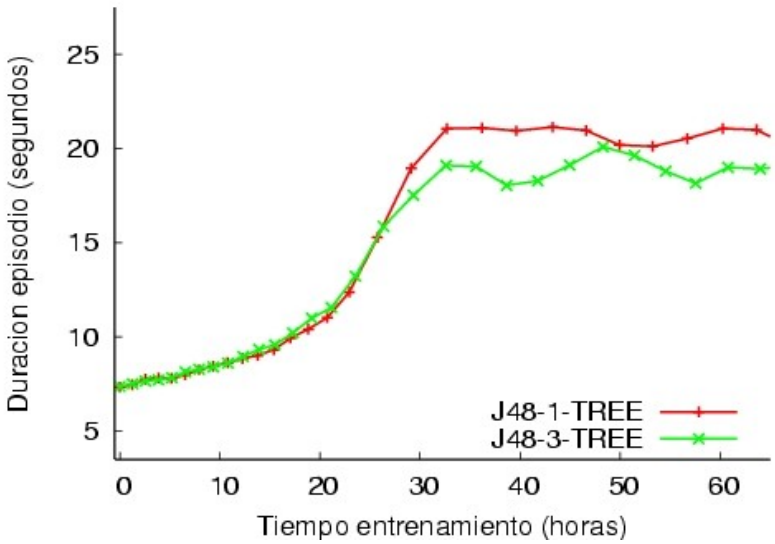

Figura C-1: Comparativa J48 1-Tree vs 3-Trees.

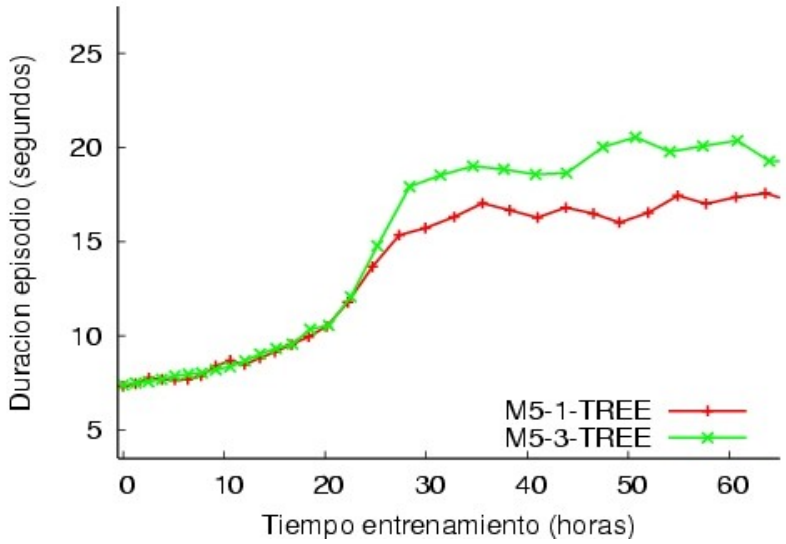

Figura C-2: Comparativa M5 1-Tree vs 3-Trees.

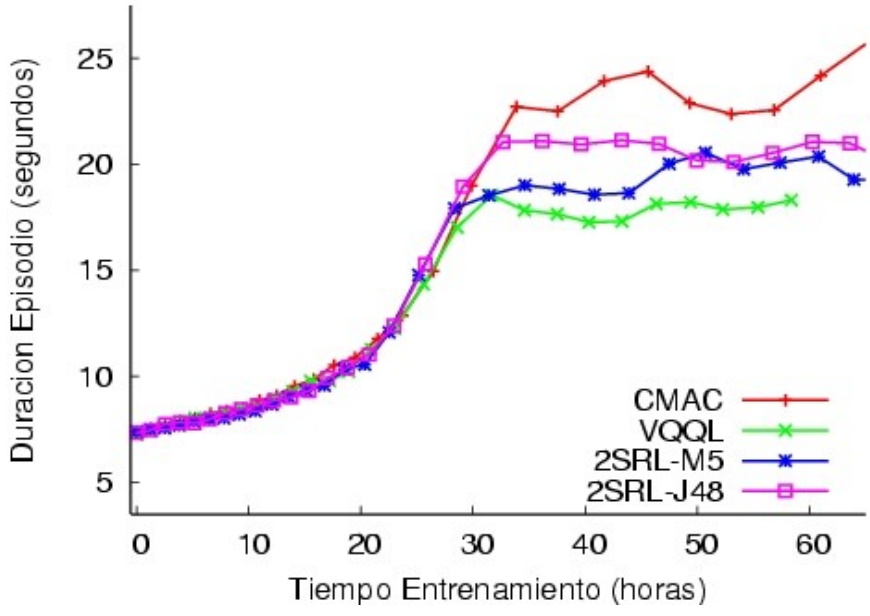

Figura C-3: Comparativa 2SRL-J48 vs 2SRL-M5 vs VQQL vs CMAC.

5 Conclusiones y trabajo futuro

En este apartado se intenta resumir a grandes rasgos las conclusiones que se pueden extraer de los resultados obtenidos en la fase de experimentación. Además se proponen nuevas líneas de investigación en las que este proyecto no ha profundizado.

5.1 Conclusiones

En el presente proyecto se ha probado la viabilidad del algoritmo *Two Steps Reinforcement learning* (2SRL) en la *Robocup-Soccer Keepaway*, el cual es un dominio complejo y con refuerzo continuo. Además se ha probado que es factible y pueden obtenerse buenos resultados tanto con 1 aproximador, que incorpore la acción como un parámetro más de entrada, como con *N* aproximadores, cada uno de ellos dedicado en exclusiva a una sola acción. También se ha comprobado la viabilidad de utilizar árboles de regresión en la primera fase de 2SRL, como es el caso de M5. Para comprobar lo anteriormente expuesto ha sido necesaria una ardua experimentación con todos los factores anteriormente enumerados para poder obtener resultados que corroboraran tales afirmaciones.

Se ha de tener en cuenta que la evaluación de los resultados de 2SRL requiere de un prolongado periodo de tiempo. Para completar la primera fase de una ejecución de 2SRL pueden ser necesarias hasta 20 horas, dependiendo de los parámetros configurados y el número de iteraciones de ISQL que se deseen obtener. Si a eso se le añade que cada ejecución del simulador necesita de al menos 15 horas en un equipo de última generación para obtener una simulación completa de la *keepaway* que muestre la evolución de la curva de aprendizaje, el tiempo total para evaluar un solo experimento completo de 2SRL puede alargarse más de una semana, dependiendo de la profundidad de detalle de la evaluación.

Con respecto a las configuraciones probadas de 2SRL comprobamos un mejor rendimiento del algoritmo cuando utilizamos un clasificador J48, siempre que la discretización del espacio de recompensas sea adecuada. Utilizar una discretización del espacio de recompensas adecuada es determinante para conseguir una mejor política. Esto supone un gran trabajo de experimentación para dar con esta discretización adecuada, ya que es necesario realizar la discretización manualmente. Para solventar este problema se han realizado experimentos con M5, el cuál genera un árbol de regresión. La principal ventaja de M5 frente a J48, es que el primero no necesita discretizar el espacio de recompensas o refuerzos a estimar, ya que funciona con valores continuos. De esta manera eliminamos una de las variables de configuración más tediosa. Los resultados obtenidos con M5 son lo suficientemente satisfactorios como para plantearlo como alternativa al uso de J48 u otro algoritmo de clasificación.

Realizando la comparativa con modelos como CMAC y VQQL se ha demostrado que 2SRL es una técnica que ofrece resultados a la altura de estas técnicas, generando buenas políticas de acción.

Otro dato importante es que 2SRL ha funcionado en un dominio complejo como la *keepaway*, donde el aprendizaje lo realizan de manera simultanea varios agentes que persiguen un objetivo común. Aunque cada agente aprende de manera individual y no recibe información de los demás agentes más allá de la que el entorno le transmite a través de sus sensores, cada agente aprende una política que hace que el objetivo común de todos los agentes se cumpla. En este caso el objetivo global es mantener el balón el máximo tiempo posible en posesión del equipo de los *keepers*.

Como aportación adicional de este proyecto, algunos de los resultados obtenidos han sido publicados en un congreso [López-bueno et al, 2009].

5.1 Trabajo futuro

Hay varios puntos que este trabajo no ha abordado. El primero de ellos es comprobar el rendimiento con otros escenarios de juego de *Keepaway*. En este proyecto solo se ha tratado el caso concreto de 3vs2 en un terreno de Juego de dimensiones 25x25. Sería interesante ver los resultados que se obtienen mediante 2SRL en escenarios con otro número de jugadores como, por ejemplo, 4vs3 ó 5vs4. También sería interesante comprobar el rendimiento con restricciones de visión o algún grado de error añadido en los valores captados en los sensores para hacer más real el entorno simulado. *Keepaway* permite añadir esos márgenes de error.

Las discretizaciones obtenidas solo se han probado con el algoritmo de aprendizaje por refuerzo *Q-Learning* y su variante *Q(λ)*. Sería interesante ver los resultados que obtienen otros algoritmos como, por ejemplo, *Sarsa* y *Sarsa(λ)*.

Keepaway es una subtarea de la *Robocup-Soccer*. Sería interesante probar 2SRL en problemas más cercanos al problema general del *Soccer* utilizando equipos más grandes como 11vs11 o tratando otros ámbitos del juego. Esta línea sería interesante aplicar 2SRL al aprendizaje de los *Takers* o tratar el tema de la política de movimiento para conseguir posiciones del agente óptimas para el pase. Como último paso, se podría tratar el problema completo del partido de fútbol.

En los experimentos realizados los *takers* tienen comportamientos preprogramados y todos los *keepers* aprenden simultáneamente. Se podrían abordar otros escenarios en este sentido. Como ejemplo, se podrían utilizar dos *Keepers* expertos y un *Keeper* que aprenda, o tratar el caso el el que *keepers* y *takers* aprenden simultáneamente durante el juego.

A. Implementación de ISQL

En el presente apartado se expone la implementación del algoritmo ISQL orientado a ser utilizado con los ficheros de tuplas generados en el entorno de la *keepaway*. Estos ficheros contienen tuplas con el formato descrito en el apartado 2.4. También estos ficheros tendrán parámetros especiales y respetarán el formato de fichero ARFF de WEKA [Frank y Witten, 2005]. más adelante se verá con más detalle el formato de los fichero de entrada que aceptará la presente implementación del ISQL. Principalmente este apartado contiene los diagramas UML y la descripción de clases de los módulos que componen la implementación de ISQL, la forma en que estos módulos interactuan y la descripción del formato de los ficheros de entrada y salida de cada módulo.

Como queda descrito en el apartado 3.2 el diseño modular de ISQL se compone de los siguientes módulos:

- Actualizador
- Generador
- Parser
- Compilador

Los módulos “Generador” y “Compilador” se refieren a Weka y al compilador C/C++ respectivamente. Estas herramientas son ampliamente conocidas y sus diseños e implementaciones no corresponde ser descritas en el presente proyecto. En cambio, los módulos “Actualizador” y “Parser” han sido implementados exclusivamente para el presente proyecto y su implementación se describirá en los siguientes apartados.

A.1 El módulo Actualizador.

Como se describió en el apartado 3.1, el módulo actualizador se encarga de leer un fichero de tuplas y aplicar la ecuación de actualización de Q-Learning a los refuerzos de cada tupla, generando un fichero con las tuplas actualizadas. Este módulo se ejecutará una vez por cada iteración que realice ISQL. Al a la hora de implementar el algoritmo habrá ligeras diferencias si lo implementamos para 1 aproximador que incluye la acción como parámetro o para N aproximadores, cada uno para una acción distinta.

A.1.1 Implementación para 1 aproximador

El diagrama UML que describe el conjunto de clases que compone este módulo se muestra en la figura A.1-1.

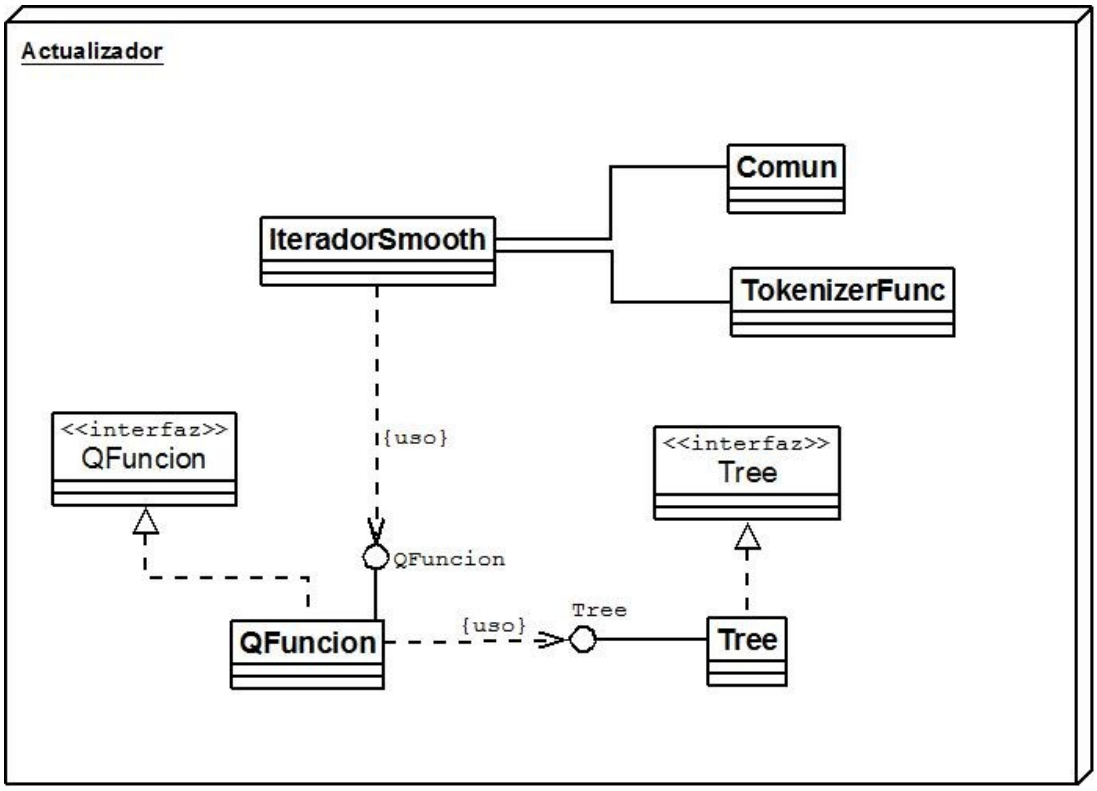

Figura A.1-1: Diagrama UML del Actualizador 1 Aproximador.

Esta implementación respeta el diseño visto en el apartado 3.1. Su idea principal es conseguir cambiar de forma sencilla la implementación de la clase *Tree* sin que el resto de clases sufran ningún impacto a nivel de implementación. Para ello la clase *Qfuncion* hace de enlace entre *IteradorSmooth* y *Tree* haciendo que IteradorSmooth quede aislado de cualquier cambio que pueda sufrir la clase *Tree* tanto en su implementación como en su interfaz. Si la interfaz de *Tree* tuviera que sufrir algún cambio solo afectaría a *Qfuncion* y IteradorSmooth seguiría sin sufrir impacto alguno ya que solo interactúa con la interfaz de *Qfuncion*. Este diseño tiene otra ventaja añadida, y es que podríamos sustituir *Tree* por N clases *TreeACC_K,* haciendo este nombre alusión a que la clase implementa el aproximador para la acción K (ACC_K). Como se verá más adelante el esquema es lo suficientemente sólido para que sufra un impacto mínimo en la adaptación de la implementación a N aproximadores.

La clase Comun

La clase Comun implementa algunas de las funciones de uso común en el proceso de actualización de tuplas. A continuación se detallan los atributos y los métodos de la clase.

Tipo	Atributo	Descripción
Definición de tipo (struct en .h)	Atributo	Definición de la estructura atributo compuesta por char value[], char name[], char type[], int used.
Definición de tipo (struct en .h)	ParseAtributo	Definición de la estructura ParseAtributo compuesta por int intValue, double doubleValue.

Tabla 34: Atributos de la clase Comun.

Método	
Nombre	parse
Parámetros	Atributo* at, ParseAtributo* val
retorno	void
descripción	Realiza una conversión de tipo del valor del atributo al que apunta el puntero at. El valor del atributo está siempre en forma de cadena de caracteres (at.value) y será guardado en la estructura apuntada por val con el tipo original definido (at.type).
visibilidad	pública

Método	
Nombre	classValue
Parámetros	char* _class
retorno	double
descripción	Devuelve el valor númerico correspondiente a la cadena _class. Las cadenas _class tendrán la forma “clase_Valor”. Así para la cadena “clase_30” el valor devuelto será un double 30.00
visibilidad	pública

Tabla 35: Métodos de la clase comun.

La clase TokenizerFunc

La clase TokenizerFunc implementa algunas funciones que facilitan el tratamiento de cadenas de caracteres y ficheros de texto plano. Su función es poder tratar una cadena o fichero como fragmentos o campos separados por caracteres delimitadores. A continuación se detallan los atributos y los métodos de la clase TokenizerFunc.

Método	
Nombre	getToken
Parámetros	char* tok,char delimit[],int delimit_length,FILE *FICHERO
retorno	int
descripción	Escribe en el buffer al que apunta tok el siguiente token del fichero FICHERO teniendo como criterio de separación los caracteres delimitadores contenidos en el array delimit de tamaño delimit_length. El entero que devuelve indica el numero de caracteres recuperados en tok. Cuando la función no encuentra más token en el fichero el puntero de fichero se posiciona en EOF
visibilidad	pública

Método	
Nombre	getTokenStr
Parámetros	char* tok,char delimit[],int delimit_length,char* string,int ini
retorno	int
descripción	Escribe en el buffer al que apunta tok el siguiente token de la cadena string a partir de la posición de desplazamiento ini teniendo como criterio de separación los caracteres delimitadores contenidos en el array delimit de tamaño delimit_length. El entero que devuelve indica el ultimo caracter procesado tok.
visibilidad	pública

Método	
Nombre	noBound
Parámetros	char c,char delimit[],int delimit_length
retorno	int
descripción	Dado un caracter c y un array de caracteres delimit la función devuelve -1 si c se encuentra entre los caracteres de delimit. Si el caracter no se encuentra devuelve 1.
visibilidad	privada

Tabla 36: Métodos de la clase TokenizerFunc.

La interfaz Qfuncion y la clase Qfuncion

La interfaz Qfuncion define los métodos que IteradorSmooth necesita para realizar las actualizaciones de los refuerzos. De este modo IteradorSmooth no utiliza directamente los aproximadores/clasificadores sino que pide el resultado a los métodos de Qfuncion y éstos calculan el resultado utilizando el aproximador implementado por la clase Tree.

Los métodos que define la interfaz Qfuncion y que implementa la clase Qfuncion se muestran a continuación.

Método	
Nombre	Qfun_max
Parámetros	double state[],int numKeepers,int numTakers
retorno	double
descripción	Función que dado el estado actual (state) y el número de keepers (numKeepers) y Takers (numTakers) en el juego, devuelve el mayor valor obtenido al ejecutar todas las posibles acciones en el estado actual.
visibilidad	pública

Método	
Nombre	Qfun
Parámetros	double state[],int action
retorno	double
descripción	Función que dado el estado actual (state) y la acción a ejecutar (action) devuelve el refuerzo esperado.
visibilidad	pública

Tabla 37: Métodos de la clase Qfuncion.

La interfaz Tree y la clase Tree

La interfaz Tree define el método que toda clase o librería que implemente un aproximador debe tener. Esta interfaz define como se accede a los aproximadores y que parámetros necesita para obtener una estimación. Los métodos que define la interfaz Tree y que debe implementar la clase Tree se muestran a continuación.

Método	
Nombre	tree
Parámetros	double state[],double reward,int action
retorno	double
descripción	Función que dado el estado actual (state) y la acción a ejecutar (action) devuelve el refuerzo esperado. El parámetro reward no es necesario y no es utilizado, solo es necesario proporcionar un valor cualquiera como por ejemplo -1.
visibilidad	pública

Tabla 38: Métodos de la clase Tree.

La clase IteradorSmooth

Esta clase implementa todo el procedimiento de lectura del fichero de tuplas actualización y escritura de las tuplas actualizadas. Contiene el método principal del módulo Actualizador. Los métodos que componen la clase se muestran a continuación.

Método	
Nombre	procesarFichero
Parámetros	FILE *fichero,int iter
retorno	int
descripción	Función que realiza el procesamiento del fichero de tuplas “fichero” en la iteración “iter” del algoritmo ISQL
visibilidad	privada

Método	
Nombre	main
Parámetros	int argc,char* argv[]
retorno	int
descripción	Función principal que ejecuta el resto de procedimientos necesarios para que funcione el Actualizador. Sus argumentos se reciben por línea de comandos. Devuelve 0 cuando la condición de parada del algoritmo se cumple. Si no se cumple la condición de parada de ISQL devuelve 1.
visibilidad	pública

Tabla 39: Métodos de la clase IteradorSmooth.

A.1.2 Implementación para N aproximadores

Como se aludió en el anterior apartado, adaptar el anterior esquema para que funcione con varios aproximadores es muy sencillo. Se pueden reutilizar las clases IteradorSmooth, Comun y TokenizerFunc. La clase Qfuncion deberá ser ligeramente modificada en su implementación. El diagrama UML que describe el conjunto de clases que compone este módulo modificado para tener tantos aproximadores como acciones posibles se muestra en la figura A.1-2. En concreto esta figura muestra el diseño para 3 acciones.

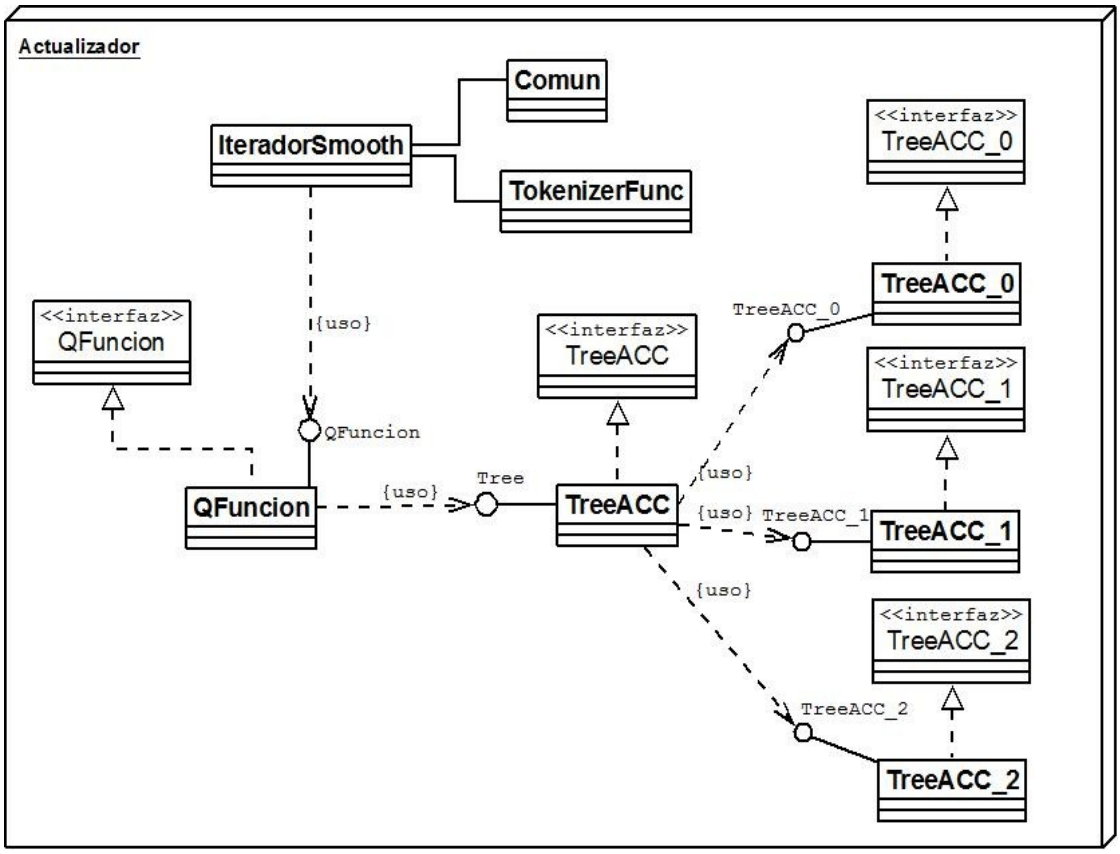

Figura A.1-2: Diagrama UML del Actualizador 3 Aproximadores.

En el diagrama anterior se ha sustituido la clase e interfaz Tree por la clase e interfaz TreeACC que distribuye las peticiones de estimación entre los distintos aproximadores por acción. Cuando, desde Qfuncion, se invoca la estimación de un refuerzo para el estado actual y una acción i, la petición se delega al estimador i desde TreeACC. En este apartado solo se describirán las clases nuevas que no aparecían en el apartado anterior. La implementación mostrada es la realizada para ser utilizada con tuplas de un entorno de la *keepaway* 3vs2.

La clase TreeACC

La clase TreeACC recibe las peticiones que hace Qfuncion. De este modo Qfuncion no es consciente de cuantos aproximadores existen para resolver las estimaciones de la función Q. TreeACC se encarga de llamar al estimador correcto según la acción recibida. Los métodos que componen la interfaz y, en consecuencia, la clase TreeACC se muestran a continuación.

Método	
Nombre	treeACC
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado-acción. Recibe como parámetros el estado actual y la acción a ejecutar. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = action. El parámetro reward no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 40: Métodos de la clase TreeACC.

Las clases TreeACC_K

Cada clase TreeACC_K implementa el estimador de refuerzos para la acción k. Se define una interfaz para cada estimador para que la implementación sea posible en lenguajes que no estén orientados a objetos y que no acepten el repetir el nombre de procedimiento en tiempo de compilación. Para nuestro caso concreto, en un entorno 3vs2 de la keepaway, se implementarán tres interfaces: TreeACC_0, TreeACC_1 y TreeACC_2. Los métodos de estas clases son se definen a continuación.

Clase TreeACC_0

Implementación del aproximador de la función Q para la acción 0 (Hold).

Método	
Nombre	treeACC_0
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado para la acción 0. Recibe como parámetro el estado actual. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = 0. El parámetro reward y la acción no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 41: Métodos de la clase TreeACC_0.

Clase TreeACC_1

Implementación del aproximador de la función Q para la acción 1 (Pass keeper 1).

Método	
Nombre	treeACC_1
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado para la acción 1. Recibe como parámetro el estado actual. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = 1. El parámetro reward y la acción no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 42: Métodos de la clase TreeACC_1.

Clase TreeACC_2

Implementación del aproximador de la función Q para la acción 2 (Pass keeper 2).

Método	
Nombre	treeACC_2
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado para la acción 2. Recibe como parámetro el estado actual. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = 2. El parámetro reward y la acción no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 43: Métodos de la clase TreeACC_2.

A.2 El módulo Parser.

El módulo Parser es una pieza clave a la hora de poder automatizar el proceso iterativo que demanda ISQL. Este módulo software transforma un modelo proporcionado por una herramienta externa, como es Weka, en una clase o librería compilable en lenguaje c/c++ que cumple las propiedades definidas por el modelo original. La figura A.2-1 muestra el diagrama UML del módulo Parser.

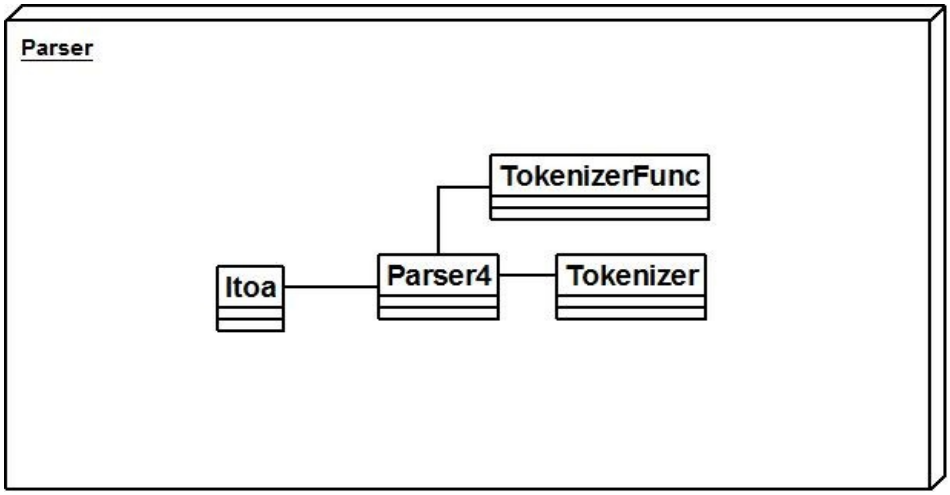

Figura A.2-1: Diagrama UML del Parser.

Sobre esta implementación hay que realizar varias aclaraciones. En primer lugar, la implementación está realizada en c/c++ y este lenguaje permite la declaración de estructuras de tipo registro (struct). Estas estructuras, que aquí son recogidas con el mismo formato que cualquier atributo, deberían traducirse en clases en un diseño UML orientado a objetos. Si así se hiciera, en el diagrama anterior aparecería una clase para cada estructura representando el tipo que pretende representar la estructura con sus métodos de acceso a atributos (getters y setters). Como ya se ha mencionado las definiciones de estructuras serán recogidas como declaración de atributos, siendo conscientes de que no es el mismo concepto, aunque aquí aparezcan en la misma sección y con el mismo formato que los atributos simples.

Otro punto a aclarar es la diferencia entre el Parser utilizado para el modelo con un solo aproximador y el de varios. Prácticamente la implementación es idéntica y por ese motivo solo se especifican las clases del Parser con 1 aproximador. La principal diferencia entre ellos reside simplemente en que la versión con varios aproximadores realiza un renombrado de los ficheros de salida añadiendo el sufijo de la acción a la que corresponde el aproximador de salida. Esto se hace para permitir la convivencia y la identificación de los distintos aproximadores dentro del modelo.

La clase TokenizerFunc

La clase TokenizerFunc implementa algunas funciones que facilitan el tratamiento de cadenas de caracteres y ficheros de texto plano. Su función es poder tratar una cadena o fichero como fragmentos o campos separados por caracteres delimitadores. A continuación se detallan los atributos y los métodos de la clase TokenizerFunc.

Método	
Nombre	getToken
Parámetros	char* tok,char delimit[],int delimit_length,FILE *FICHERO
retorno	int
descripción	Escribe en el buffer al que apunta tok el siguiente token del fichero FICHERO teniendo como criterio de separación los caracteres delimitadores contenidos en el array delimit de tamaño delimit_length. El entero que devuelve indica el numero de caracteres recuperados en tok. Cuando la función no encuentra más token en el fichero el puntero de fichero se posiciona en EOF
visibilidad	pública

Método	
Nombre	getTokenStr
Parámetros	char* tok,char delimit[],int delimit_length,char* string,int ini
retorno	int
descripción	Escribe en el buffer al que apunta tok el siguiente token de la cadena string a partir de la posición de desplazamiento ini teniendo como criterio de separación los caracteres delimitadores contenidos en el array delimit de tamaño delimit_length. El entero que devuelve indica el ultimo caracter procesado tok.
visibilidad	pública

Método	
Nombre	noBound
Parámetros	char c,char delimit[],int delimit_length
retorno	int
descripción	Dado un caracter c y un array de caracteres delimit la función devuelve -1 si c se encuentra entre los caracteres de delimit. Si el caracter no se encuentra devuelve 1.
visibilidad	privada

Tabla 44: Métodos de la clase TokenizerFunc.

La clase Tokenizer

La clase Tokenizer implementa algunas funciones que facilitan el tratamiento de ficheros de texto plano. Su función es poder tratar un fichero como fragmentos o campos separados por caracteres delimitadores. A continuación se detallan los atributos y los métodos de la clase Tokenizer. Tokenizer está implementada con las directrices de c++.

Tipo	Atributo	Descripción
FILE* (puntero)	FICHERO	Descriptor de fichero que apunta al fichero de texto plano del que se obtendrán los tokens.
char[]	nombreFichero	Cadena de caracteres que representa al nombre del fichero.
char[]	delimit	Array de caracteres que contiene los caracteres que serán usados como delimitadores de tokens
int	delimit_length	Longitud del array delimit

Tabla 45: Atributos de la clase Tokenizer.

Método	
Nombre	Tokenizer
Parámetros	char* fich,char* bound,int range
retorno	No aplicable
descripción	Constructor de la clase Tokenizer. Necesita para su construcción un nombre de fichero fich, un array de caracteres delimitadores bound y un rango que medirá el número de caracteres delimitadores que contiene bound
visibilidad	pública

Método	
Nombre	open
Parámetros	void
retorno	FILE* (puntero)
descripción	Intenta abrir el fichero especificado en el atributo nombreFichero. Si lo consigue devuelve el descriptor del fichero abierto.
visibilidad	pública

Método	
Nombre	getToken
Parámetros	char* tok (puntero)
retorno	int
descripción	Intenta copiar el siguiente token encontrado en el fichero al buffer apuntado por tok. El entero que devuelve es el número de caracteres recuperados en tok.
visibilidad	pública

Método	
Nombre	eof
Parámetros	void
retorno	int
descripción	Devuelve 0 si el puntero a fichero abierto no está en el final. Devuelve un número mayor de 0 de lo contrario.
visibilidad	pública

Método	
Nombre	close
Parámetros	void
retorno	int
descripción	Intenta cerrar el fichero abierto. Si tiene éxito devuelve 1 si no devuelve un valor 0 o negativo.
visibilidad	pública

Método	
Nombre	noBound
Parámetros	char c
retorno	int
descripción	Dado un caracter c, la función devuelve -1 si c se encuentra entre los caracteres del atributo delimit. Si el caracter no se encuentra devuelve 1.
visibilidad	privada

Tabla 46: Métodos de la clase Tokenizer.

La clase Parser4

La clase Parser implementa la lógica que permite realizar la conversión del fichero del modelo de regresión/clasificación a código compilable en c/c++. El sufijo 4 añadido al nombre alude a que es la cuarta versión implementada del Parser, ya que se ha esta clase ha sufrido un constante refinamiento para tratar toda la casuística del formato del modelo proporcionado a la salida estándar de Weka. Más adelante, en el apartado referente al formato de los ficheros de entrada de cada módulo se especificará con más detalle los formatos contemplados en esta cuarta versión de la implementación del Parser. Los atributos, estructuras y métodos definidos en esta clase se muestran a continuación.

Tipo	Atributo	Descripción
Definición de tipo (struct en .h)	Symbol	Definición de la estructura Symbol compuesta por char token[] (el símbolo), char tab[] (tipo de tratamiento que se hará al símbolo), char type[] (si está tipado, tipo del símbolo), int Tsymbol_pos (posición relativa en la tabla de símbolos).
Definición de tipo (struct en .h)	TSymbol	Definición de la estructura TSymbol compuesta por char translate[], char tab[], char type[].
Definición de tipo (struct en .h)	TildeClass	Definición de la estructura TildeClass compuesta por char classname[], int used.
int	num_atrr	Número de atributos y a su vez símbolos que aparecerán en el modelo a traducir, que son atributos discriminadores en la clasificación ó traducciones de símbolos (consultar apartado de ficheros de entrada de Parser).
Symbol [ ]	stable	Array de elementos Symbol que compone la tabla de símbolos manejados por el parser.
TSymbol [ ]	tclasses	Array de elementos Tsymbol que compone la tabla de clases que contiene las diferentes clases en las que se pueden clasificar las entradas del modelo.
Char [ ]	delimitadores	Array de caracteres que contienen los caracteres delimitadores estándar para el tratamiento de ficheros y cadenas como contenedores de tokens.
int	d_length	Número de caracteres contenidos en el array “delimitadores”.
Char [ ]	tree_name	Nombre del fichero de salida que contendrá el modelo parseado.

Char [ ]	tree_name_h	Nombre del fichero de cabecera correspondiente al fichero de salida “tree_name”
Char [ ]	tree_temp	Nombre del fichero que será usado para un tratamiento intermedio necesario para la traducción modelo → fichero_compilable.
int	NUM_HOJAS	Contador del número de hojas que tiene el árbol que se está traduciendo.

Tabla 47: Atributos de la clase Parser4.

Método	
Nombre	replace
Parámetros	char* replaced,char* buf
retorno	void
descripción	Busca el token apuntado por “buf” en la tabla de símbolos. Si es encontrado en la tabla de símbolos se hará la traducción correspondiente del símbolo y el resultado de la traducción se depositará en el buffer al que apunta “replaced”. Si no se encuentra en la tabla se copia el mismo token en “replaced”.
visibilidad	privada

Método	
Nombre	isSymbol
Parámetros	char *str,char *t
retorno	int
descripción	Devuelve distinto de 0 si el la cadena a la que apunta “str” es un estado del tipo indicado en la cadena a al que apunta “t”.
visibilidad	privada

Método	
Nombre	isClass
Parámetros	char* inclass
retorno	int
descripción	Devuelve 0 si la cadena apuntada por “inclass” es un nombre de alguna clase registrada en la tabla de clases.
visibilidad	privada

Método	
Nombre	isLM
Parámetros	char* inclass
retorno	int
descripción	Devuelve 0 si “inclass” es una cabecera de modelo lineal (LM) con formato de modelo de regresión de Weka M5P.
visibilidad	privada

Método	
Nombre	tratar_condicion_m5
Parámetros	char* replaced,char* buf,FILE* modelo
retorno	void
descripción	Trata la condición del modelo M5 al que apunta el descriptor “modelo” cuando “buf” apunta al primer token de la condición, dejando la condición en formato compilable de c/c++ en “replaced.
visibilidad	privada

Método	
Nombre	make_if_m5
Parámetros	FILE* MODEL,char* buf,FILE* FICH,int profant
retorno	int
descripción	Trata el conjunto de reglas del modelo al que apunta “MODEL” y las transforma en condiciones if-else compilables en c/c++, escribiendo el modelo completamente traducido en el fichero al que apunta “FICH”. Profant guarda el nivel de anidamiento de la llamada ya que el método funciona de forma recursiva.
visibilidad	privada

Método	
Nombre	make_LM
Parámetros	FILE* MODEL,char* buf,FILE* FICH
retorno	void
descripción	Trata el conjunto de modelos lineales definidos en el fichero “MODEL” y transforma cada modelo lineal en una función compilable e invocable en c/c++.
visibilidad	privada

Método	
Nombre	tratar_condicion_J48
Parámetros	char* replaced,char* buf,FILE* modelo
retorno	int
descripción	Trata la condición del modelo J48 al que apunta el descriptor “modelo” cuando “buf” apunta al primer token de la condición, dejando la condición en formato compilable de c/c++ en “replaced”. El entero que devuelve indica el tipo de condición tratada.
visibilidad	privada

Método	
Nombre	classValue
Parámetros	char* _class
retorno	double
descripción	Devuelve el valor numérico que se corresponde con la clase a la que apunta “_class”
visibilidad	privada

Método	
Nombre	make_Class
Parámetros	FILE* FICH
retorno	void
descripción	Escribe en el fichero “FICH” una función compilable en c/c++ por cada clase registrada en la tabla de clases que devuelve el valor de la clase.
visibilidad	privada

Método	
Nombre	make_if_J48
Parámetros	FILE* MODEL,char* buf,FILE* FICH,int profant
retorno	int
descripción	Trata el conjunto de reglas del modelo al que apunta “MODEL” y las transforma en condiciones if-else compilables en c/c++, escribiendo el modelo completamente traducido en el fichero al que apunta “FICH”. Profant guarda el nivel de anidamiento de la llamada ya que el método funciona de forma recursiva.
visibilidad	privada

Método	
Nombre	main
Parámetros	int argc, char* argv[]
retorno	int
descripción	Método principal que utiliza el resto de funciones de la clase para conseguir crear un modelo compilable en c/c++. Los parámetros se rellenan con los argumentos pasados por línea de comandos.
visibilidad	publica

Tabla 48: Métodos de la clase Parser4.

A.3 Orquestación de los módulos.

En este apartado se describirá como los distintos módulos interactuan para llevar a cabo cada una de las iteraciones de ISQL. El proceso funciona como una cadena en donde cada módulo espera los ficheros de entrada que necesita, que son salidas de un módulo anterior. Las invocaciones de cada módulo las hará un shell-script que irá llamando a los módulos en el orden adecuado. Según el script va llamado a los módulos estos van dejando sus ficheros de salida en el directorio de trabajo, en donde los siguientes módulos irán a buscar esos ficheros como entrada. A continuación se analizarán el shell-script para un aproximador y para N aproximadores.

A.3.1 shell-script para 1 aproximador

En este apartado se describe el script *iterar.sh* que orquesta la ejecución de los módulos del ISQL con un aproximador. El script se puede dividir en 3 partes:

- Configuración: donde se definen los valores de las variables que deben ser definidas en función del entorno en el que deseemos ejecutar el algoritmo.
- Inicio: conjunto de operaciones que deja preparado el entorno de ejecución de ISQL para poder realizar el procesamiento automático.
- Procesamiento: Región del script que define el bucle iterativo de ISQL.

En el apartado de configuración se definen las siguientes variables:

Variable	Valor
ARCHIVO	Ruta del archivo de tuplas inicial.
ALGORITMO	Algoritmo de Weka que se aplicará a las tuplas actualizadas en cada iteración para generar un modelo.
ENTRENAMIENTO	Nombre del fichero de tuplas actualizadas que servirá como entrenamiento para el algoritmo de Weka.
MODELO	Nombre del fichero del modelo de salida de Weka.
PARSERCONFIG	Nombre del fichero de configuración que utilizará el Parser para generar el modelo. Este fichero de configuración se genera como salida del módulo actualizador.
KEEPAWAY	Ruta donde está instalada la keepaway.
WEKA	Ruta donde está instalado Weka.
DIR_RESULTS	Ruta en donde se guardarán los ficheros de salida de cada iteración del algoritmo. Los fichero que se guardan son los de tuplas actualizadas y los modelos parseados.

Tabla 49: Variables de configuración del script iterar.sh.

En la figura A.3-1 podemos ver una captura de la sección de variables descrita.

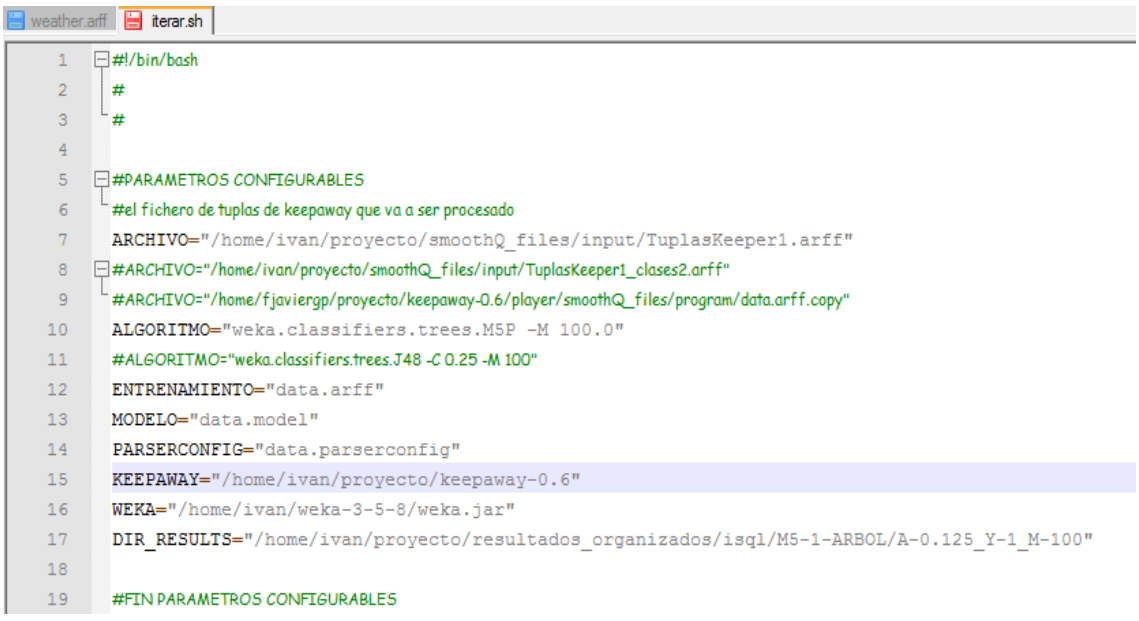

Figura A.3-1: Sección de variables del script iterar.sh.

La sección de inicio o inicialización se ejecuta cuando el script recibe como parámetro la cadena “ini”. Al ejecutarse con este parámetro el algoritmo realiza la primera compilación de los módulos y prepara los ficheros necesarios para iterar. El concreto el proceso es el siguiente:

1. Se copia el fichero del aproximador inicial al directorio de compilación del módulo actualizador.
2. Se compila el actualizador (IteradorSmooth).
3. Se ejecuta el actualizador que recibe como parámetros “ARCHIVO” y el número de iteración, en este caso 0.
4. Se hace una copia del fichero “ENTRENAMIENTO” generado en la ejecución del actualizador (extensión “.copy”) que servirá en el futuro como tabla Q. También se guarda una copia del fichero “ENTRENAMIENTO” en el directorio “DIR_RESULTS”.
5. Se llama a Weka con los parámetros “ALGORITMO” y “ENTRENAMIENTO” generando el fichero “MODELO” como salida.
6. Se llama al parser con los parámetros “PARSERCONFIG” que tratará el archivo “MODELO” y generará un fichero escrito en lenguaje C.
7. Se copia el archivo C al directorio del actualizador y se guarda una copia en “DIR_RESULTS.
8. Se vuelve a compilar el actualizador con el nuevo fichero C que contiene en nuevo aproximador.

Con este proceso que consigue realizar la iteración 0 del algoritmo, cuya principal diferencia con las otras iteraciones es que necesita compilar el actualizador con el primer aproximador, que simplemente devuelve 0 (no aproxima). La sección del script *iterar.sh* que realiza la inicialización puede observarse en la figura A.3-2.

```
55  if [ $INI = "ini" ] ; then
56
57      echo INICIALIZANDO ALGORITMO
58      echo "-----------------------"
59      echo "copiando arbol inicial a iterador"
60      cp iterador/Tree_inicial/* iterador
61      cd iterador
62      compilar.sh
63      cd ..
64      echo "Procesando fichero $ARCHIVO de keepaway"
65      IteradorSmooth $ARCHIVO 0
66      echo "Generado archivo data.arff para procesamiento weka"
67      cp data.arff data.arff.copy
68      echo "Copiando tuplas de la iteracion a dir resultados"
69      cp data.arff "$DIR_RESULTS/data.0.arff"
70      #si se entra aquí entonces el iterador ha generado nuevos resultados para procesar
71      echo "Generando modelo $ALGORITMO -x 2 -t > $MODELO ..."
72      java -Xms1500m -Xmx2200m -cp $WEKA $ALGORITMO -x 2 -t $ENTRENAMIENTO>$MODELO
73      echo "Parseando modelo $MODELO"
74      ParserV4 $PARSERCONFIG -h
75      echo "Copiando arbol parseado a dir resultados"
76      cp -f Tree.c "$DIR_RESULTS/Tree.0.c"
77      echo "moviendo arbol generado a iterador"
78      mv Tree.* iterador
79      cd iterador
80      compilar.sh
81      cd ..
```

Figura A.3-2:Sección de inicialización del script iterar.sh.

La sección de procesamiento orquesta el buque de que define cada iteración del ISQL. La descripción del proceso es el siguiente:

1. Se inicia el bucle iterativo que se repetirá mientras la variable “SEGUR” valga 1.
2. Se ejecuta el actualizador que recibe como parámetros “ARCHIVO” y el número de iteración.
3. Si al ejecutar el módulo actualizador el resultado indica que se debe parar (por criterio de proximidad entre tuplas de la iteración anterior y actual) entonces “SEGUIR” tomará valor 0 y el bucle finaliza. Si el iterador (actualizador) indica que se debe seguir iterando se pasa al punto 4.
4. Se hace una copia del fichero “ENTRENAMIENTO” generado en la ejecución del actualizador (extensión “.copy”) que servirá en el futuro como tabla Q. También se guarda una copia del fichero “ENTRENAMIENTO” en el directorio “DIR_RESULTS”.
5. Se llama a Weka con los parámetros “ALGORITMO” y “ENTRENAMIENTO” generando el fichero “MODELO” como salida.
6. Se llama al parser con los parámetros “PARSERCONFIG” que tratará el archivo “MODELO” y generará un fichero escrito en lenguaje C.
7. Se copia el archivo C al directorio del actualizador y se guarda una copia en “DIR_RESULTS.

8. Se vuelve a compilar el actualizador con el nuevo fichero C que contiene en nuevo aproximador.
9. Se incrementa el contador e iteraciones y se vuelve a 2.

La sección del script *iterar.sh* que realiza el procesamiento puede observarse en la figura A.3-3.

```
137     while test $SEGUIR -eq 1 ; do
138
139         echo "\$ITERACION: $ITERACION"
140         echo "------------------------"
141         echo "Procesando fichero $ARCHIVO de keepaway"
142         if IteradorSmooth $ARCHIVO $ITERACION ; then
143             echo "Generando archivo data.arff para procesamiento weka"
144             cp data.arff data.arff.copy
145             echo "Copiando tuplas de la iteracion a dir resultados"
146             cp -f data.arff "$DIR_RESULTS/data.$ITERACION.arff"
147             #si se entra aquí entonces el iterador ha generado nuevos resultados para procesar
148             echo "Generando modelo $ALGORITMO -x 2 -t > $MODELO ..."
149             java -Xms1500m -Xmx2200m -cp $WEKA $ALGORITMO -x 2 -t $ENTRENAMIENTO>$MODELO
150             echo "Parseando modelo $MODELO"
151             ParserV4 $PARSERCONFIG -h
152             echo "Copiando arbol parseado a dir resultados"
153             cp -f Tree.c "$DIR_RESULTS/Tree.$ITERACION.c"
154             echo "moviendo arbol generado a iterador"
155             mv Tree.* iterador
156             cd iterador
157             compilar.sh
158             cd ..
159         else
160
161             SEGUIR=0
162
163         fi
164         let ITERACION=$ITERACION+1
165     done
```

Figura A.3-3: Sección de procesamiento del script iterar.sh.

A.3.2 shell-script para N aproximadores

En este apartado se describe el script *iterarACC.sh* que orquesta la ejecución de los módulos del ISQL con N aproximadores, uno por acción. El modo en que se plantea es muy similar al descrito para 1 solo aproximador pero adaptado para manejar e integrar varios aproximadores. Como el proyecto se ha centrado en la configuración 3vs2 es para esta configuración para la que se ha implementado el algoritmo, es decir, disponemos de 3 aproximadores: uno para la acción 0 (hold), otro para la acción 1 (pass Keeper 1) y otro para la acción 2 (pass keeper 2). el script también se puede dividir en configuración, inicio y procesamiento, pero está vez el procesamiento y el inicio puede ser subdividido en otras tres secciones:

- Inicio/Procesamiento para la acción 0
- Inicio/Procesamiento para la acción 1
- Inicio/Procesamiento para la acción 2

Como se puede deducir, estas tres fases son idénticas para cada acción y podría ser introducida en un buque, aunque en este caso tan sencillo con tan pocas acciones se ha optado por desplegar el código para mayor claridad.

En el apartado de configuración se definen las siguientes variables:

Variable	Valor
ARCHIVO_0	Ruta del archivo de tuplas inicial para la acción 0.
ARCHIVO_1	Ruta del archivo de tuplas inicial para la acción 1.
ARCHIVO_2	Ruta del archivo de tuplas inicial para la acción 2.
PODA=0.25 MIN_X_HOJA=100 OPS_ALG=""	Estos tres parámetros son de configuración del algoritmo y complementan la información de la variable "ALGORITMO". En el script para un solo aproximador visto anteriormente esta información estaba incluida directamente en la variable que define el algoritmo.
ALGORITMO	Algoritmo de Weka que se aplicará a las tuplas actualizadas en cada iteración para generar un modelo.
ENTRENAMIENTO	Nombre del fichero de tuplas actualizadas que servirá como entrenamiento para el algoritmo de Weka.
MODELO	Nombre del fichero del modelo de salida de Weka.
PARSERCONFIG	Nombre del fichero de configuración que utilizará el Parser para generar el modelo. Este fichero de configuración se genera como salida del módulo actualizador.
KEEPAWAY	Ruta donde está instalada la keepaway.
WEKA	Ruta donde está instalado Weka.
DIR_RESULTS	Ruta en donde se guardarán los ficheros de salida de cada iteración del algoritmo. Los fichero que se guardan son los de tuplas actualizadas y los modelos parseados.

Tabla 50: Variables de configuración del script IterarACC.sh.

Se describe a continuación la inicialización:

1. Se copian los ficheros de los aproximadores iniciales al directorio de compilación del módulo actualizador .
2. Se compila el actualizador (IteradorSmoothACC).
3. Se ejecuta el actualizador que recibe como parámetros "ARCHIVO_0" y el número de iteración, en este caso 0.

4. Se hace una copia del fichero “ENTRENAMIENTO” generado en la ejecución del actualizador (extensión “.copy”) que servirá en el futuro como tabla Q. También se guarda una copia del fichero “ENTRENAMIENTO” en el directorio “DIR_RESULTS”.
5. Se llama a Weka con los parámetros “ALGORITMO” y “ENTRENAMIENTO” generando el fichero “MODELO” como salida.
6. Se llama al parser con los parámetros “PARSERCONFIG” que tratará el archivo “MODELO” y generará un fichero escrito en lenguaje C.
7. Se copia el archivo C al directorio del actualizador y se guarda una copia en “DIR_RESULTS.
8. Se repite el proceso desde 3 hasta 7 para las acciones 1 (ARCHIVO_1) y 2 (ARCHIVO_2).
9. Se vuelve a compilar el actualizador con el los nuevos ficheros C que contienen cada uno un nuevo aproximador.

La sección de procesamiento orquesta el buque de que define cada iteración del ISQL. La descripción del proceso con N aproximadores es el siguiente:

1. Se inicia el bucle iterativo que se repetirá mientras la variable “SEGUR” valga 1.
2. Se ejecuta el actualizador que recibe como parámetros “ARCHIVO_0” y el número de iteración.
3. Si al ejecutar el módulo actualizador el resultado indica que se debe parar (por criterio de proximidad entre tuplas de la iteración anterior y actual) entonces “SEGUIR_0” tomará valor 0. Si el iterador (actualizador) indica que se debe seguir iterando se pasa al punto 4.
4. Se hace una copia del fichero “ENTRENAMIENTO” generado en la ejecución del actualizador (extensión “.copy”) que servirá en el futuro como tabla Q. También se guarda una copia del fichero “ENTRENAMIENTO” en el directorio “DIR_RESULTS”.
5. Se llama a Weka con los parámetros “ALGORITMO” y “ENTRENAMIENTO” generando el fichero “MODELO” como salida.
6. Se llama al parser con los parámetros “PARSERCONFIG” que tratará el archivo “MODELO” y generará un fichero escrito en lenguaje C.
7. Se copia el archivo C al directorio del actualizador y se guarda una copia en “DIR_RESULTS.
8. Se repite el proceso desde 2 hasta 7 para las acciones 1 (ARCHIVO_1) y 2 (ARCHIVO_2).
9. Se vuelve a compilar el actualizador con los nuevos ficheros C que contienen los nuevos aproximadores.
10. Se incrementa el contador e iteraciones y se vuelve a 2.

A.4 Formato de los ficheros de entrada y salida de cada módulo.

En este apartado de detallan los ficheros de entrada y salida que se generan en la ejecución de cada módulo y el formato que respeta cada fichero.

A.4.1 Ficheros relacionados con el módulo Actualizador

Los ficheros de entrada para el actualizador son:

- Fichero inicial de tuplas en formato “ARFF” con parámetros adicionales de configuración del algoritmo ISQL.
- Fichero “copia” del fichero de tuplas actualizadas de la iteración anterior del algoritmo ISQL.

Los ficheros de salida que genera el actualizador son:

- Fichero de tuplas actualizadas.
- Fichero con extensión “.parserconfig” que determina la configuración correcta del parser en esa iteración para ese conjunto de tuplas.
- Fichero log de distancia entre los refuerzos de las tuplas de la iteración anterior y la actual.

A continuación se profundiza en el formato de cada fichero.

Formato del Fichero inicial de tuplas.

El formato definido para este fichero es muy similar al utilizado por Weka en sus ficheros “ARFF” con algunos cambios. Al principio del fichero se definen los siguientes elementos (El formato de esto elementos comienzan siempre por %etiqueta valor) :

- %mode: Modo de funcionamiento. Hace alusión al tipo de clasificador que será utilizado en las actualizaciones. Sus valores soportados son weka.classifiers.trees.J48 y weka.classifiers.trees.M5P.
- %numkeepers: Número de keepers del entorno con el que se han generado las tuplas de experiencia.
- %numtakers: Número de takers del entorno con el que se han generado las tuplas de experiencia.
- %error: distancia mínima entre los refuerzos de las tuplas de la iteración actual y la anterior que sirve como criterio de parada del algoritmo.
- %alpha: Constante α que será utilizada en la ecuación de actualización de los refuerzos.
- %ypsilon: Constante Υ que será utilizada en la ecuación de actualización de los refuerzos.

El resto de los elementos respetan el formato “arff” y definen los atributos del estado, acción, estado siguiente, refuerzo y si la tupla es de fin de episodio o no. Se puede ver un ejemplo de este tipo de fichero en la figura A.4-1.

```
%fichero de keepaway-keeper para IteradorSmooth
%mode weka.classifiers.trees.J48
%numkeepers 3
%numtakers 2
%error 1000.01
%alpha 0.125
%ypsilon 1
@relation keepaway-keeper
@attribute wb_dist_to_c real
@attribute wb_dist_to_k1 real
@attribute wb_dist_to_k2 real
@attribute wb_dist_to_t0 real
@attribute wb_dist_to_t1 real
@attribute dist_to_c_k1 real
@attribute dist_to_c_k2 real
@attribute dist_to_c_t0 real
@attribute dist_to_c_t1 real
@attribute nearest_opp_dist_k1 real
@attribute nearest_opp_dist_k2 real
@attribute nearest_opp_ang_k1 real
@attribute nearest_opp_ang_k2 real
@attribute action {0,1,2}
@attribute next_wb_dist_to_c real
@attribute next_wb_dist_to_k1 real
@attribute next_wb_dist_to_k2 real
@attribute next_wb_dist_to_t0 real
@attribute next_wb_dist_to_t1 real
@attribute next_dist_to_c_k1 real
@attribute next_dist_to_c_k2 real
@attribute next_dist_to_c_t0 real
```

Figura A.4-1: Cabecera de fichero de tuplas de entrada del Actualizador.

Formato del Fichero de tuplas actualizadas

El fichero de tuplas actualizadas que se genera como salida del actualizador sigue el formato “ARFF” de Weka. En este fichero se definen los atributos que definen el estado actual, la acción y el refuerzo recibido. Además la cabecera inicial indica la iteración en la que se ha generado el fichero. Este fichero sigue un formato que permite ser pasado directamente a la entrada de Weka para que sea utilizado por el algoritmo de clasificación adecuado para generar un aproximador de la función de refuerzo. La figura A.4-2 muestra un extracto de un fichero de tuplas generado por ISQL en la iteración 0 para J48.

```
%fichero weka de iteracion 0 smoothQLearning
%numkeepers 3
%numtakers 2
@relation keepaway-keeper
@attribute wb_dist_to_c real
@attribute wb_dist_to_k1 real
@attribute wb_dist_to_k2 real
@attribute wb_dist_to_t0 real
@attribute wb_dist_to_t1 real
@attribute dist_to_c_k1 real
@attribute dist_to_c_k2 real
@attribute dist_to_c_t0 real
@attribute dist_to_c_t1 real
@attribute nearest_opp_dist_k1 real
@attribute nearest_opp_dist_k2 real
@attribute nearest_opp_ang_k1 real
@attribute nearest_opp_ang_k2 real
@attribute action {0,1,2}
@attribute reward {clase0 clase1 clase2 clase3 clase4 clase5 clase6 clase7}
@data

10.415463,8.2,18.2,10,11,8.348103,11.426918,3.155331,6.575485,3.850765,14.566377,16,53,0,clase0
12.76201,18.2,27.1,16.4,18.2,12.859043,14.341318,12.328706,15.09907,25.128519,20.499079,93,49,0,clase0
12.435345,18.2,24.5,14.9,16.4,13.023333,12.066412,11.412574,14.050981,24.312201,18.527582,94,49,0,clase0
```

Figura A.4-2: Cabecera de fichero de tuplas actualizadas.

Formato del Fichero de configuración del parser ".parserconfig"

Este fichero es generado de forma automática por el actualizador para mantener la coherencia entre el número de clases, que puede crecer en el proceso dinámico de actualización de tuplas, y el número de clases con el que tiene que trabajar el parser para traducir el modelo. Su formato es muy sencillo y muy parecido a la declaración de atributos del formato "ARFF" con la diferencia de que este fichero solo contiene declaraciones y no instancias. En la figura A.4-3 podemos ver un ejemplo de fichero de configuración que utilizaremos para definir el formato general.

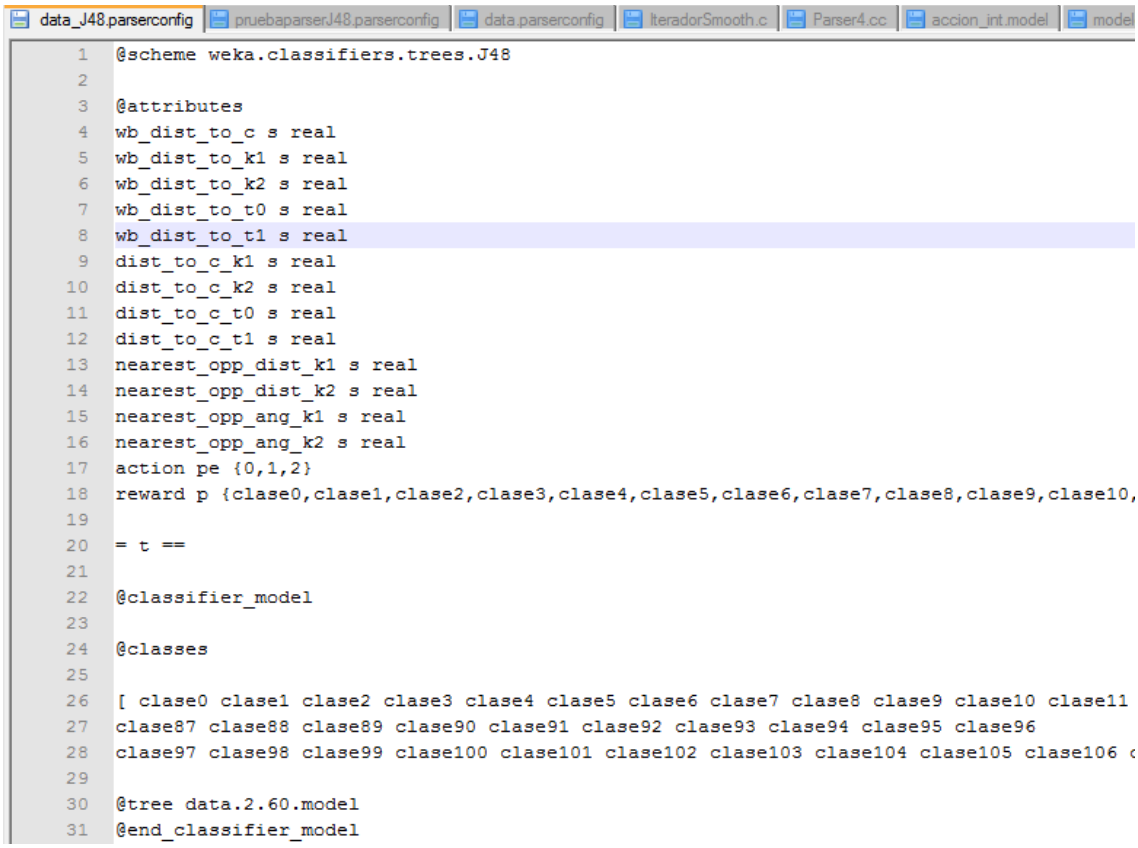

Figura A.4-3: Ejemplo de fichero de configuración del parser.

El fichero de configuración se divide en tres secciones etiquetadas por una palabra que define su inicio:

- @scheme: identifica el esquema que sigue el modelo del fichero de entrada. El parser puede soportar dos tipos de modelos: weka.classifiers.trees.J48 y weka.classifiers.trees.M5P.
- @attributes: a partir de esta etiqueta se definen por cada línea un atributo que aparecerá en el fichero del modelo. Estos atributos corresponden con los atributos manejados por el clasificador para generar el modelo que estamos parseado. El formato de cada línea debe respetar la definición <nombre_atributo> espacio <tratamiento del atributo> espacio <tipo del atributo>. Un ejemplo de definición puede ser la siguiente: [wb_dist_to_c s real]. En nombre simplemente define el atributo, el tipo soporta los mismos tipos que los definidos en los ficheros “ARFF” y el tratamiento soporta varias opciones:

    - s → Indica al parser que este atributo es parte del conjunto que define un estado de la keepaway. Debe ser traducido a en el modelo al elemento correspondiente del array que define los estados en el simulador (state[i]).

- t → Indica al parser que este atributo debe ser traducido al ser encontrado en el modelo. Por ejemplo si encontramos un signo de igualdad en el modelo, para la correcta compilación del fichero parseado debemos traducir el signo '=' a '==' (= t ==).
- p → Indica que el atributo no forma parte del conjunto que define un estado pero es una entrada del aproximador. Un ejemplo de este tipo de atributo es la acción.
- pe → Indica lo mismo que el atributo p con la diferencia de que se especifica que el tipo a tratar es enumerado y puede encontrarse en el modelo expresado de forma especial.

■ @classifier_model: Esta sección se define el modelo. Si el esquema es para J48 tiene dos subsecciones:
  - @classes: Define el conjunto de clases con el que está definido el modelo.
  - @tree: Define la ruta del fichero en el que el parser tiene que buscar el modelo porporcionado por Weka.

  En caso utilizar el esquema M5P esta sección solo contiene la subsección @tree. El fin de esta sección viene determinada por la etiqueta "@end_classifer_model".

A.4.2 Ficheros relacionados con el módulo Generador

Cuando hablamos de módulo generador nos referimos a Weka. Para que Weka pueda generar un modelo necesita como entrada un conjunto de tuplas en formato "ARFF". El fichero que se utiliza como entrenamiento es el fichero de tuplas actualizadas que genera el actualizador en cada iteración, cuyo formato se ha definido en el apartado A.4.1. Como salida, Weka genera un fichero que contiene un modelo escrito de en un formato que define la lógica con la que se comporta el modelo. El formato del modelo se puede observar en el apartado referente a Weka.

A.4.3 Ficheros relacionados con el módulo Parser

Los ficheros de entrada para el parser son:

■ El fichero ".parserconfig" de configuración generado por el actualizador y cuyo formato ha sido definido en el apartado A.4.1.
■ El fichero con el modelo proporcionado como salida de la ejecución del generador (Weka).

Los ficheros de salida para el parser son:

■ El fichero ".c" que implementa el modelo de entrada y que respeta la interfaz "Tree" (o "treeACC_K", según corresponda) definida en el apartado A.1.

- El fichero de cabecera “.h” si es requerido por los parámetros de entrada de la ejecución del parser.
- El fichero “tam.log” que registra el número de regiones que define el modelo parseado.

A.4.4 Ficheros relacionados con el módulo Compilador

El Compilador de C/C++ recibe como entrada los fichero fijos del actualizador junto con los ficheros que implementan los modelos parseados generados en la ejecución del parser. Como salida, el compilador proporciona el fichero ejecutable del módulo actualizador.

B. Implementación de la arquitectura de evaluación

En este apartado se describe la implementación de los agentes que servirán como banco de pruebas de los estimadores generados con el algoritmo ISQL. Este entorno de pruebas incluye la implementación de la segunda fase del algoritmo 2SRL, conocida como MDQL. En primer lugar se especifica el procedimiento genérico para implementar un agente que pueda interactuar con el entorno de la keepaway. En segundo y tercer lugar se describe la implementación del entorno de pruebas para las fases ISQL y MDQL respectivamente.

B.1 Implementación de un agente de la keepaway

Este apartado resume el procedimiento necesario para incluir un nuevo controlador de agente en la keepaway que guíe la conducta de los keepers. Además, no solo podemos limitarnos a implementar el control del agente, si no que dentro del controlador podemos implementar la lógica necesaria para implementar un algoritmo de aprendizaje como Q-Learning.

En el entorno de la keepaway, cada agente debe implementar un interfaz denominada “SMDPAgent”. Esta interfaz consta de tres simples métodos que permiten tratar la keepaway como un SMDP y aplicar algoritmos de aprendizaje durante la ejecución de los agentes de una forma sencilla. Los tres métodos que componen la interfaz “SMDPAgent” son:

- int startEpisode(double state[]). Este método es invocado por el agente la primera vez que recibe la pelota en el episodio y debe decidir qué acción realizará la primera vez: mantener la posesión de la pelota o pasarla a algún compañero. Si durante la ejecución de un episodio, el jugador nunca obtiene la pelota, este método nunca será invocado. Para la ejecución de este método el agente recibe un array que representa el estado en el que se encuentra el agente al recibir la pelota y se espera que devuelva la acción que ha decidido realizar en su codificación entera (K=0 - hold, K>0 - pasar a k).
- int step(double reward, double state[]). Este método es invocado por el agente cada vez que recibe la pelota después de la primera recepción. Para la ejecución de esté método el agente recibe el refuerzo (número de pasos del simulador que han transcurrido desde la última acción que ejecutó) y el estado actual en el que se encuentra. La invocación de este método devuelve la acción que el agente ha decidido realizar.
- void endEpisode(double reward). Esté método se ejecuta cuando el jugador recibe una notificación del servidor informándole de que el episodio ha finalizado. Al agente, en la invocación al método, se le pasa el refuerzo acumulado hasta la finalización del episodio. Esta función es

ejecutada después de cada episodio aunque el jugador no haya estado en posesión de la pelota durante ningún momento del mismo.

Estos tres métodos son los que permitirán la comunicación entre el controlador-agente y el entorno. Por supuesto, se pueden definir otras funciones de apoyo, pero siempre teniendo en cuenta de que los únicos métodos invocados serán los de la interfaz “SMDPAgent”.

B.2 Implementación de la arquitectura de evaluación de ISQL

El diagrama de clases de define la implementación del entorno de pruebas de los aproximadores generados en el algoritmo ISQL se define en la figura B.2-1.

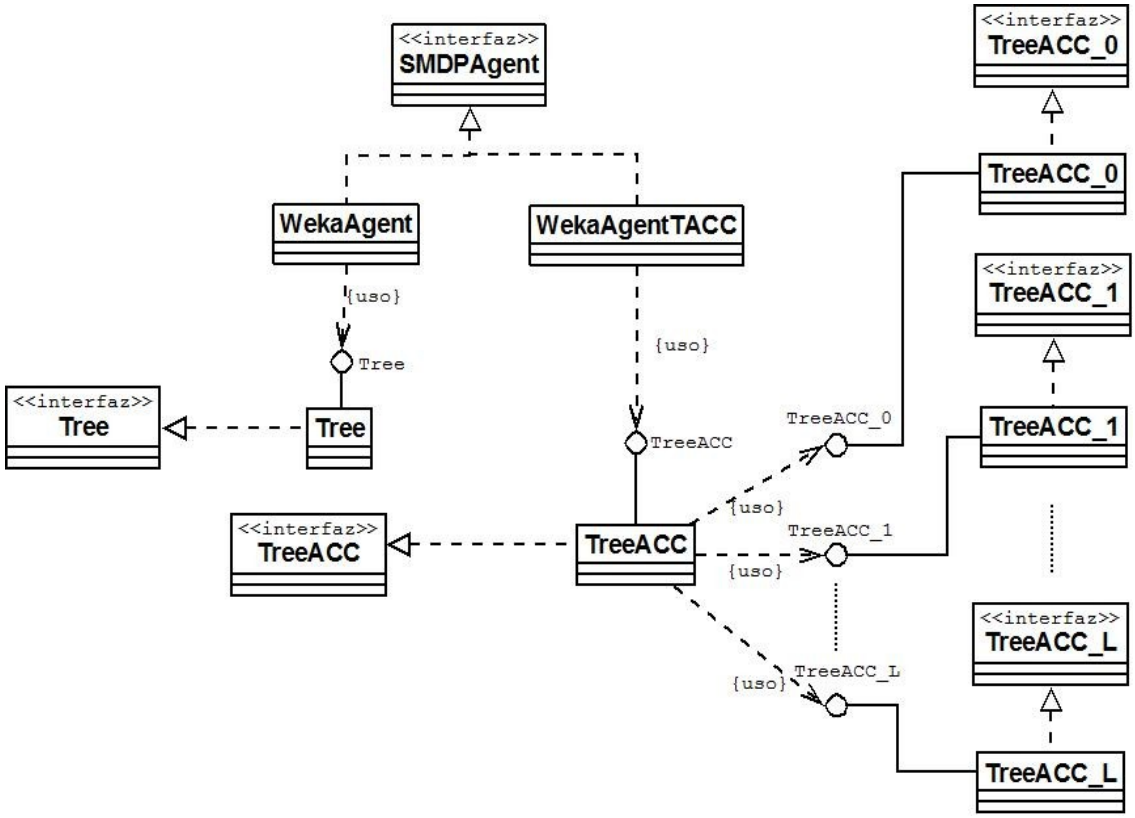

Figura B.2-1: Diagrama UML de la implementación de agentes ISQL para un entorno con L-1 keepers (L acciones).

El diagrama anterior define la implementación generalizada para cualquier configuración de juego, siendo L el número de acciones posibles (y por tanto L-1 el número de keepers en el juego). Los agentes implementados son “WekaAgent” y “WekaAgentTACC”. “WekaAgent” implementa el agente que prueba los aproximadores que incluyen la acción como parámetro para estimar el refuerzo, es decir, el uso de un solo aproximador. En cambió

“WekaAgentTACC” implementa el agente que utiliza L aproximadores, uno para cada acción, para decidir que acción ejecutar. En este esquema los aproximadores tiene como misión generar una estimación del refuerzo que se recibirá en el futuro. Está estimación será utilizada para la toma de decisiones. Como se puede observar en el diagrama UML, ambos controladores implementan la interfaz “SMDPAgent” necesaria para poder interactuar con el entorno.

Cada uno de los agentes accede a los aproximadores con una interfaz distinta. La interfaz “Tree” se define para el uso de un solo aproximador para todas las acciones y “TreeACC” es utilizada cuando se quiere utilizar un aproximador para cada acción. La figura B.2-2 define el diagrama UML para un entorno 3vs2, que es el utilizado en este proyecto.

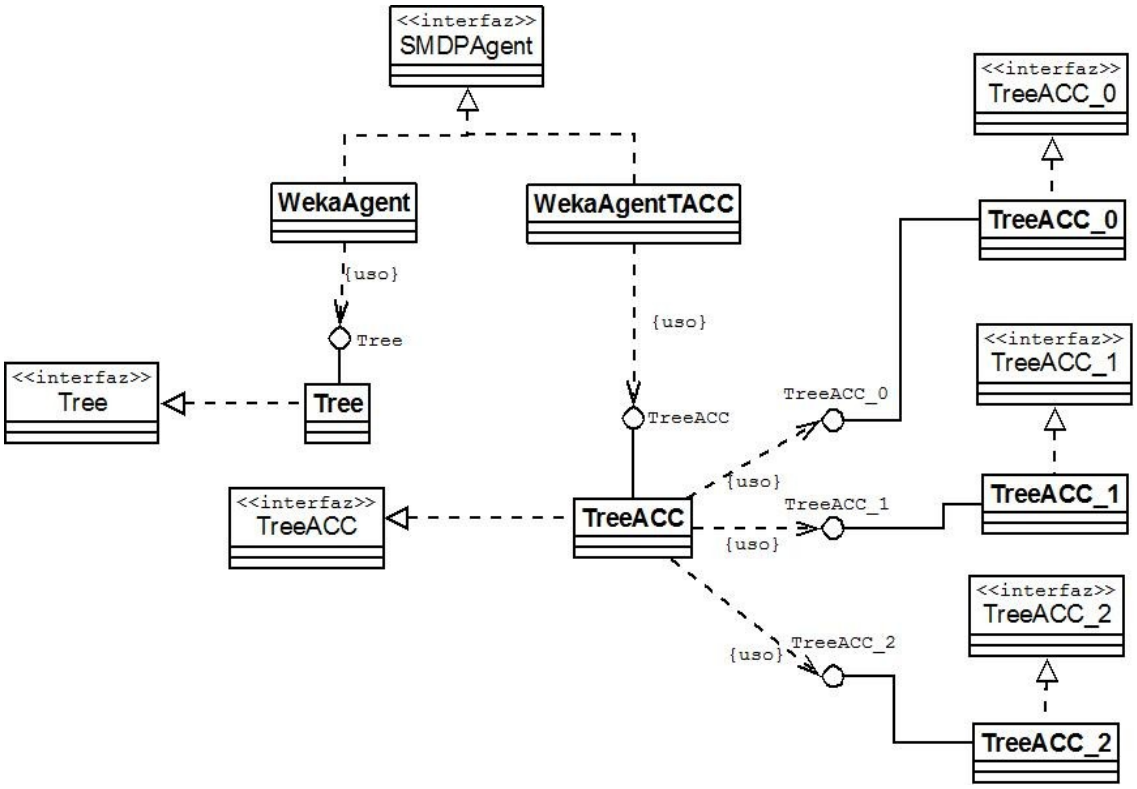

Figura B.2-2: Diagrama UML de la implementación de agentes ISQL para un entorno 3vs2.

La clase WekaAgent

La clase WekaAgent implementa la interfaz SMDPAgent y a su vez el controlador que utiliza un aproximador para determinar la recompensa futura que recibirá una acción en un estado concreto, y así, poder realiza la toma de decisiones teniendo como criterio esta recompensa estimada. Los atributos, estructuras y métodos de esta clase de definen a continuación.

Tipo	Atributo	Descripción
WorldModel* (puntero)	WM	Puntero al modelo del mundo que proporciona información sobre el entorno en el que juega el agente, como por ejemplo el número de keepers y takers que están jugando.
FILE*	FICHERO	Puntero a fichero que es utilizado para volcar trazas de ejecución en forma de tuplas.
int	abierto	Indicador de si el fichero al que apunta FICHERO ha sido abierto
char [ ]	policy	Política utilizada por el controlador en caso de que el controlador implemente varias políticas de acción
int	OPCODE	Código entero que determina el nivel de detalle de trazado de la ejecución por salida std.
int	firstrew	Indica si es la primera vez que se ejecuta step o es la última del episodio antes de empezar el siguiente episodio.

Tabla 51: Atributos de la clase WekaAgent.

Método	
Nombre	choice
Parámetros	double state[],double reward
retorno	int
descripción	Devuelve la acción que mejor previsión de refuerzo ha obtenido para el estado “state”.
visibilidad	privada

Método	
Nombre	startEpisode
Parámetros	double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón por primera vez en el episodio actual. Recibe el estado actual y devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	step
Parámetros	double reward, double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón en el episodio actual excepto la primera vez. Recibe el estado actual y el tiempo que ha pasado desde que se deshizo del balón y lo ha vuelto a recuperar. Devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	endEpisode
Parámetros	double reward
retorno	void
descripción	Se ejecuta cuando termina un episodio. Recibe el tiempo que ha pasado desde que se deshizo del balón y el momento en el que el episodio finaliza.
visibilidad	pública

Tabla 52: Métodos de la clase WekaAgent.

La clase e interfaz Tree

La clase Tree implementa el aproximador, que realiza estimaciones del refuerzo que se recibirá. También Implementa la interfaz “Tree” que permite a la clase WekaAgent poder hacer uso del estimador. El método de interfaz que se tiene que implementar se define a continuación.

Método	
Nombre	tree
Parámetros	double state[],double reward,int action
retorno	double
descripción	Dados state y action, devuelve la previsión de refuerzo que se recibirá. El parámetro reward no se utiliza y puede invocarse la función con el valor de reward que se prefiera, ya que no influye en la previsión. Por ejemplo, reward = -1.
visibilidad	pública

Tabla 53: Método de interfaz de la clase Tree.

La clase WekaAgentTACC

La clase WekaAgentTACC implementa la interfaz SMDPAgent y a su vez el controlador que utiliza tantos aproximadores como acciones para determinar la recompensa futura que recibirá una acción en un estado concreto, y así, poder realiza la toma de decisiones teniendo como criterio esta recompensa estimada. Los atributos, estructuras y métodos de esta clase de definen a continuación.

Tipo	Atributo	Descripción
WorldModel* (puntero)	WM	Puntero al modelo del mundo que proporciona información sobre el entorno en el que juega el agente, como por ejemplo el número de keepers y takers que están jugando.
FILE*	FICHERO	Puntero a fichero que es utilizado para volcar trazas de ejecución en forma de tuplas.
int	abierto	Indicador de si el fichero al que apunta FICHERO ha sido abierto
char [ ]	policy	Política utilizada por el controlador en caso de que el controlador implemente varias políticas de acción
int	OPCODE	Código entero que determina el nivel de detalle de trazado de la ejecución por salida std.
int	firstrew	Indica si es la primera vez que se ejecuta step o es la última del episodio antes de empezar el siguiente episodio.

Tabla 54: Atributos de la clase WekaAgentTACC.

Método	
Nombre	choice
Parámetros	double state[],double reward
retorno	int
descripción	Devuelve la acción que mejor previsión de refuerzo ha obtenido para el estado “state”.
visibilidad	privada

Método	
Nombre	startEpisode
Parámetros	double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón por primera vez en el episodio actual. Recibe el estado actual y devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	step
Parámetros	double reward, double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón en el episodio actual excepto la primera vez. Recibe el estado actual y el tiempo que ha pasado desde que se deshizo del balón y lo ha vuelto a recuperar. Devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	endEpisode
Parámetros	double reward
retorno	void
descripción	Se ejecuta cuando termina un episodio. Recibe el tiempo que ha pasado desde que se deshizo del balón y el momento en el que el episodio finaliza.
visibilidad	pública

Tabla 55: Métodos de la clase WekaAgentTACC.

La clase e interfaz TreeACC

La clase TreeACC implementa la interfaz “TreeACC” que permite a la clase “WekaAgentTACC” poder hacer uso de los estimadores de la función Q para cada acción sin tener que preocuparse de que estimador debe usar en cada momento. La implementación de la interfaz “TreeACC” se encarga de llamar al estimador correcto según la acción recibida. Los métodos que componen la interfaz y, en consecuencia, la clase TreeACC se muestran a continuación.

Método	
Nombre	treeACC
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado-acción. Recibe como parámetros el estado actual y la acción a ejecutar. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = action. El parámetro reward no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 56: Atributos de la clase TreeACC.

Las clases TreeACC_K

Cada clase TreeACC_K implementa el estimador de refuerzos para la acción k. Se define una interfaz para cada estimador para que la implementación sea posible en lenguajes que no estén orientados a objetos y que no acepten el repetir el nombre de procedimiento en tiempo de compilación. Para nuestro caso concreto, en un entorno 3vs2 de la keepaway, se implementarán tres interfaces: TreeACC_0, TreeACC_1 y TreeACC_2. Los métodos de estas clases son se definen a continuación.

Clase TreeACC_0

Implementación del aproximador de la función Q para la acción 0 (Hold).

Método	
Nombre	treeACC_0
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado para la acción 0. Recibe como parámetro el estado actual. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = 0. El parámetro reward y la acción no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 57: Métodos de la clase TreeACC_0.

Clase TreeACC_1

Implementación del aproximador de la función Q para la acción 1 (Pass keeper 1).

Método	
Nombre	treeACC_1
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado para la acción 1. Recibe como parámetro el estado actual. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = 1. El parámetro reward y la acción no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 58: Métodos de la clase TreeACC_1.

Clase TreeACC_2

Implementación del aproximador de la función Q para la acción 2 (Pass keeper 2).

Método	
Nombre	treeACC_2
Parámetros	double state[],double reward,int action
retorno	double
descripción	Interfaz del estimador de recompensas según estado para la acción 2. Recibe como parámetro el estado actual. Devuelve la recompensa esperada cuando desde el estado recibido se ejecuta la acción = 2. El parámetro reward y la acción no es necesario. Debe rellenarse con cualquier valor pero no es tenido en cuenta para el procesamiento de la estimación.
visibilidad	pública

Tabla 59: Métodos de la clase TreeACC_2.

B.3 Implementación de la arquitectura de evaluación de MDQL

El objetivo de la arquitectura de evaluación MDQL es poder probar las distintas divisiones del espacio de estados de la keepaway que proporcionan los aproximadores generados en ISQL, utilizándolas en algoritmos de aprendizaje por refuerzo con representaciones tabulares de la función Q. En este apartado se especifica la implementación de los agentes desarrollados para tal efecto.

B.3.1 Implementación para 1 aproximador/discretizador.

Para probar las discretizaciones generadas por 1 solo árbol, que incluye como parámetro de entrada la acción, se han desarrollado 2 agentes:

- SSWekaAgent: Implementa el algoritmo de aprendizaje Q-learning con política de exploración e-greedy.
- SSWekaAgentLambda: Implementa el algoritmo Q(λ), que hace uso de trazas de elegibilidad. Al igual que el anterior agente, implementa la política de exploración e-greedy.

Ambos agentes se comunican con el árbol de clasificación a través de la interfaz TreeSS. El diagrama de clases que define la relación entre clases se puede observar en la figura B.3-1.

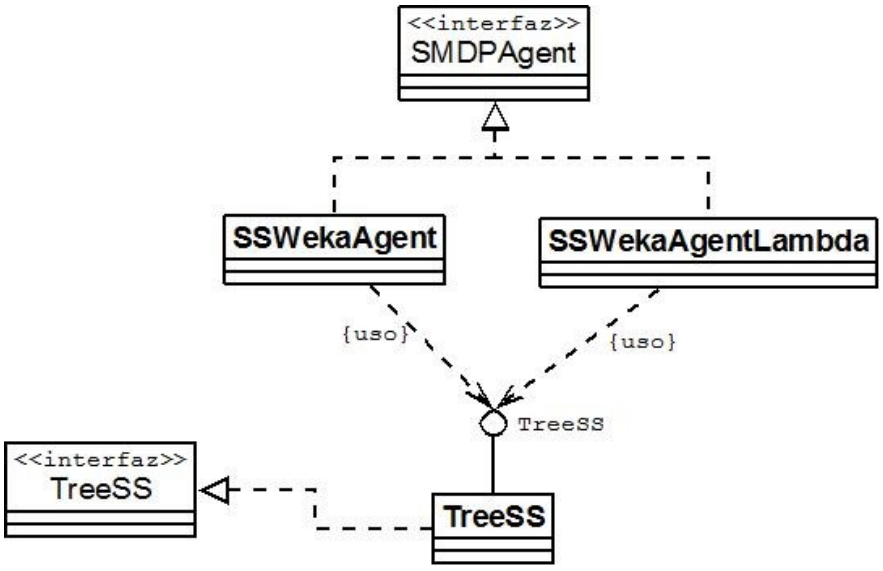

Figura B.3-1: Diagrama UML de la implementación de los agentes SSWekaAgent y SSWekaAgentLambda.

Clase SSWekaAgent

Como se ha comentado anteriormente, esta clase implementa el agente con el algoritmo de aprendizaje Q-learning. Como todo agente de la keepaway, esta clase implementa la interfaz SMDPAgent. Los atributos y métodos que la componen se presentan a continuación.

Tipo	Atributo	Descripción
WorldModel* (puntero)	WM	Puntero al modelo del mundo que proporciona información sobre el entorno en el que juega el agente, como por ejemplo el número de keepers y takers que están jugando.
FILE*	FICHERO	Puntero a fichero que es utilizado para volcar trazas de ejecución en forma de tuplas.
int	abierto	Indicador de si el fichero al que apunta FICHERO ha sido abierto
char [ ]	policy	Política utilizada por el controlador en caso de que el controlador implemente varias políticas de acción
int	OPCODE	Código entero que determina el nivel de detalle de trazado de la ejecución por salida std.
int	firstrew	Indica si es la primera vez que se ejecuta step o es la última del episodio antes de empezar el siguiente episodio.
FILE*	FICHERO_TABLA	Puntero al fichero que contiene las dimensiones y los valores de inicialización de la tabla Q.
double	CONST_A	Valor que toma la constante α en la función de actualización de la tabla Q.
double	CONST_B	Valor que toma la constante γ en la función de actualización de la tabla Q.
double	VAR_EPSILON	Valor que toma la variable ϵ en la política $\epsilon - greedy$.
double	VAR_INC_EPSILON	Valor en la que se incrementa la variable ϵ en la política $\epsilon - greedy$. El incremento se realiza al comienzo de cada episodio.
char [ ]	NOMBREQTSAVE	Nombre del fichero en el que se vuelcan los valores de la tabla Q de forma regular.
int	Q_TABLE_LENGTH	Número de celdas que tiene la tabla Q.
double* (puntero)	Q_TABLE	Puntero que apunta a la tabla Q.

int	QLEARNING	Variable que indica si se aplica el algoritmo Q-Learning sobre la tabla Q durante el juego. Q-Learning está activo solo cuanto esta variable toma el valor de 1.
int	ESCRIBIRQT	Indica cada cuantos episodios se debe volcar la tabla Q a fichero.
double	ESTADO_ANTERIOR	Puntero que apunta a la zona de memoria que guarda el estado anterior por el que transitó el agente.
int	LENGTH_ESTADO_ANTERIOR	Número de atributos que definen el estado anterior (y en consecuencia, cualquier estado del juego).
int	ACCION_ANTERIOR	Acción que el agente ejecutó en el estado anterior.
int	EPISODIOS	Número de episodios transcurridos desde el último volcado a fichero de la función Q.
int	START	Indica si el agente ha ejecutado la función startEpisode en el episodio actual.

Tabla 60: Atributos de la clase SSWekaAgent.

Método	
Nombre	choice
Parámetros	double state[],double reward
retorno	int
descripción	Devuelve la acción que mejor previsión de refuerzo ha obtenido para el estado “state”.
visibilidad	privada

Método	
Nombre	startEpisode
Parámetros	double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón por primera vez en el episodio actual. Recibe el estado actual y devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	step
Parámetros	double reward, double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón en el episodio actual excepto la primera vez. Recibe el estado actual y el tiempo que ha pasado desde que se deshizo del balón y lo ha vuelto a recuperar. Devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	endEpisode
Parámetros	double reward
retorno	void
descripción	Se ejecuta cuando termina un episodio. Recibe el tiempo que ha pasado desde que se deshizo del balón y el momento en el que el episodio finaliza.
visibilidad	pública

Método	
Nombre	choiceMAXQ
Parámetros	double state[],double reward
retorno	double
descripción	Dado el estado “state” devuelve el mayor refuerzo que se puede conseguir desde ese estado. Más en detalle, calcula el refuerzo esperado para todas las acciones y devuelve el mayor valor obtenido.
visibilidad	pública

Método	
Nombre	setEstadoAnterior
Parámetros	double state[]
retorno	void
descripción	Guarda el estado “state” en la variable ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQStep
Parámetros	double state[],double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning utilizando “state” como estado siguiente, “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQEnd
Parámetros	double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning al finalizar cada episodio, utilizando “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	escribirTablaQ
Parámetros	void
retorno	void
descripción	Vuelca a fichero los valores de la tabla Q.
visibilidad	pública

Tabla 61: Métodos de la clase SSWekaAgent.

Clase SSWekaAgentLambda

Como se ha comentado anteriormente, esta clase implementa el agente con el algoritmo de aprendizaje Q(λ), que hace uso de trazas de elegibilidad. Como todo agente de la keepaway, esta clase implementa la interfaz SMDPAgent. Los atributos y métodos que la componen se presentan a continuación.

Tipo	Atributo	Descripción
WorldModel* (puntero)	WM	Puntero al modelo del mundo que proporciona información sobre el entorno en el que juega el agente, como por ejemplo el número de keepers y takers que están jugando.
FILE*	FICHERO	Puntero a fichero que es utilizado para volcar trazas de ejecución en forma de tuplas.
int	abierto	Indicador de si el fichero al que apunta FICHERO ha sido abierto
char [ ]	policy	Política utilizada por el controlador en caso de que el controlador implemente varias políticas de acción
int	OPCODE	Código entero que determina el nivel de detalle de trazado de la ejecución por salida std.
int	firstrew	Indica si es la primera vez que se ejecuta step o es la última del episodio antes de empezar el siguiente episodio.
FILE*	FICHERO_TABLA	Puntero al fichero que contiene las dimensiones y los valores de inicialización de la tabla Q.
double	CONST_A	Valor que toma la constante α en la función de actualización de la tabla Q.
double	CONST_B	Valor que toma la constante γ en la función de actualización de la tabla Q.
double	VAR_EPSILON	Valor que toma la variable ϵ en la política $\epsilon - greedy$.
double	VAR_INC_EPSILON	Valor en la que se incrementa la variable ϵ en la política $\epsilon - greedy$. El incremento se realiza al comienzo de cada episodio.
char [ ]	NOMBREQTSAVE	Nombre del fichero en el que se vuelcan los valores de la tabla Q de forma regular.
int	Q_TABLE_LENGTH	Número de celdas que tiene la tabla Q.
double* (puntero)	Q_TABLE	Puntero que apunta a la tabla Q.
int	QLEARNING	Variable que indica si se aplica el algoritmo Q-Learning sobre la tabla Q durante el juego. Q-Learning está activo solo cuanto esta variable toma el valor de 1.
int	ESCRIBIRQT	Indica cada cuantos episodios se debe volcar la tabla Q a fichero.
double	ESTADO_ANTERI	Puntero que apunta a la zona de

	OR	memoria que guarda el estado anterior por el que transitó el agente.
int	LENGTH_ESTADO_ANTERIOR	Número de atributos que definen el estado anterior (y en consecuencia, cualquier estado del juego).
int	ACCION_ANTERIOR	Acción que el agente ejecutó en el estado anterior.
int	EPISODIOS	Número de episodios transcurridos desde el último volcado a fichero de la función Q.
int	START	Indica si el agente ha ejecutado la función startEpisode en el episodio actual.
double* (puntero)	Q_TABLE_VISITAS	Tabla de trazas de elegibilidad
double	delta	Valor delta de la función de actualización de trazas de elegibilidad.
double	elegibilidad	Valor de elegibilidad para el paso que se está tratando. Se guarda en esta variable pero se recupera originariamente de la tabla Q_TABLE_VISITAS
double	temp	Variable de uso temporal para elegibilidad.
double	f_q_ant	Valor anterior de la función Q para el paso que se está tratando.
double	nueva_elegibilidad	Variable donde se guarda temporalmente la actualización de la elegibilidad para el paso actual. Después se volcará a la tabla Q_TABLE_VISITAS.
double	lambda	Valor lambda de la función de actualización de trazas de elegibilidad.

Tabla 62: Atributos de la clase SSWekaAgentLambda.

Método	
Nombre	choice
Parámetros	double state[],double reward
retorno	int
descripción	Devuelve la acción que mejor previsión de refuerzo ha obtenido para el estado “state”.
visibilidad	privada

Método	
Nombre	startEpisode
Parámetros	double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón por primera vez en el episodio actual. Recibe el estado actual y devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	step
Parámetros	double reward, double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón en el episodio actual excepto la primera vez. Recibe el estado actual y el tiempo que ha pasado desde que se deshizo del balón y lo ha vuelto a recuperar. Devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	endEpisode
Parámetros	double reward
retorno	void
descripción	Se ejecuta cuando termina un episodio. Recibe el tiempo que ha pasado desde que se deshizo del balón y el momento en el que el episodio finaliza.
visibilidad	pública

Método	
Nombre	choiceMAXQ
Parámetros	double state[],double reward
retorno	double
descripción	Dado el estado “state” devuelve el mayor refuerzo que se puede conseguir desde ese estado. Más en detalle, calcula el refuerzo esperado para todas las acciones y devuelve el mayor valor obtenido.
visibilidad	pública

Método	
Nombre	setEstadoAnterior
Parámetros	double state[]
retorno	void
descripción	Guarda el estado “state” en la variable ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQStep
Parámetros	double state[],double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning utilizando “state” como estado siguiente, “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQEnd
Parámetros	double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning al finalizar cada episodio, utilizando “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	escribirTablaQ
Parámetros	void
retorno	void
descripción	Vuelca a fichero los valores de la tabla Q.
visibilidad	pública

Tabla 63: Métodos de la clase TreeACC_0.

La clase e interfaz TreeSS

La interfaz TreeSS define la forma en que se comunican los agentes “SSWekaAgent” y “SSWekaAgentLamda” con los discretizadores del espacio de estados. La interfaz define el siguiente método que define como deben comportarse todos los discretizadores / aproximadores.

Método	
Nombre	treeSS
Parámetros	double state[],double reward,int action
retorno	int
descripción	Devuelve el índice de la tabla Q correspondiente al par estado-acción definido por “state” y “action”.
visibilidad	pública

Tabla 64: Métodos de la clase TreeSS.

B.3.2 Implementación para 3 aproximadores/discretizadores.

Para probar las discretizaciones generadas por 3 árboles, uno por cada acción (caso 3vs2) se han desarrollado 2 agentes:

- ACCSSWekaAgent: Implementa el algoritmo de aprendizaje Q-learning con política de exploración e-greedy.
- ACCSSWekaAgentLambda: Implementa el algoritmo Q(λ), que hace uso de trazas de elegibilidad. Al igual que el anterior agente, implementa la política de exploración e-greedy.

Ambos agentes se comunican con los árboles de clasificación a través de la interfaz TreeACCSS. El diagrama de clases que define la relación entre clases se puede observar en la figura B.3-2.

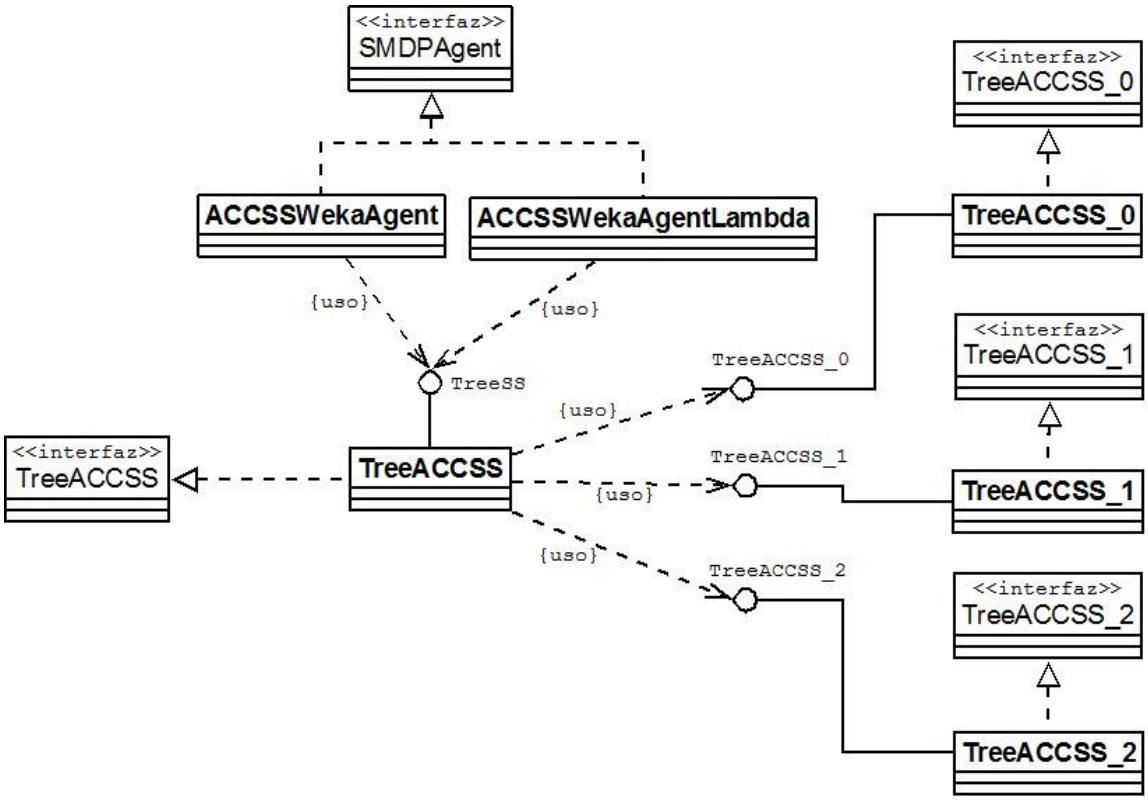

Figura B.3-2: Diagrama UML de la implementación de los agentes ACCSSWekaAgent y ACCSSWekaAgentLambda.

Clase ACCSSWekaAgent

Esta clase implementa el agente con el algoritmo de aprendizaje Q-learning utilizando una tabla Q por acción y, en consecuencia, un discretizador del espacio por cada acción. Como todo agente de la keepaway, esta clase implementa la interfaz SMDPAgent. Los atributos y métodos que la componen se presentan a continuación.

Tipo	Atributo	Descripción
WorldModel* (puntero)	WM	Puntero al modelo del mundo que proporciona información sobre el entorno en el que juega el agente, como por ejemplo el número de keepers y takers que están jugando.
FILE*	FICHERO	Puntero a fichero que es utilizado para volcar trazas de ejecución en forma de tuplas.
int	abierto	Indicador de si el fichero al que apunta FICHERO ha sido abierto

char [ ]	policy	Política utilizada por el controlador en caso de que el controlador implemente varias políticas de acción
int	OPCODE	Código entero que determina el nivel de detalle de trazado de la ejecución por salida std.
int	firstrew	Indica si es la primera vez que se ejecuta step o es la última del episodio antes de empezar el siguiente episodio.
FILE*	FICHERO_TABLA	Puntero al fichero que contiene las dimensiones y los valores de inicialización de la tabla Q.
double	CONST_A	Valor que toma la constante α en la función de actualización de la tabla Q.
double	CONST_B	Valor que toma la constante γ en la función de actualización de la tabla Q.
double	VAR_EPSILON	Valor que toma la variable ϵ en la política $\epsilon - greedy$.
double	VAR_INC_EPSILON	Valor en la que se incrementa la variable ϵ en la política $\epsilon - greedy$. El incremento se realiza al comienzo de cada episodio.
Char ** (puntero doble)	NOMBREQTSAVE	Array con el Nombre de los ficheros en el que se vuelcan los valores de las tablas Q de forma regular.
Int* (puntero)	Q_TABLE_LENGTH	Array con el número de celdas que tienen cada una de las tablas Q (una por acción).
double** (puntero doble)	Q_TABLE	Puntero que apunta a las tablas Q.
int	QLEARNING	Variable que indica si se aplica el algoritmo Q-Learning sobre la tabla Q durante el juego. Q-Learning está activo solo cuanto esta variable toma el valor de 1.
int	ESCRIBIRQT	Indica cada cuantos episodios se debe volcar la tabla Q a fichero.
double	ESTADO_ANTERIOR	Puntero que apunta a la zona de memoria que guarda el estado anterior por el que transitó el agente.
int	LENGTH_ESTADO_ANTERIOR	Número de atributos que definen el estado anterior (y en consecuencia, cualquier estado del juego).
int	ACCION_ANTERIOR	Acción que el agente ejecutó en el estado anterior.
int	EPISODIOS	Número de episodios transcurridos desde el último volcado a fichero de la función Q.

int	START	Indica si el agente ha ejecutado la función startEpisode en el episodio actual.

Tabla 65: Atributos de la clase ACCSSWekaAgent.

Método	
Nombre	choice
Parámetros	double state[],double reward
retorno	int
descripción	Devuelve la acción que mejor previsión de refuerzo ha obtenido para el estado “state”.
visibilidad	privada

Método	
Nombre	startEpisode
Parámetros	double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón por primera vez en el episodio actual. Recibe el estado actual y devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	step
Parámetros	double reward, double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón en el episodio actual excepto la primera vez. Recibe el estado actual y el tiempo que ha pasado desde que se deshizo del balón y lo ha vuelto a recuperar. Devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	endEpisode
Parámetros	double reward
retorno	void
descripción	Se ejecuta cuando termina un episodio. Recibe el tiempo que ha pasado desde que se deshizo del balón y el momento en el que el episodio finaliza.
visibilidad	pública

Método	
Nombre	choiceMAXQ
Parámetros	double state[],double reward
retorno	double
descripción	Dado el estado “state” devuelve el mayor refuerzo que se puede conseguir desde ese estado. Más en detalle, calcula el refuerzo esperado para todas las acciones y devuelve el mayor valor obtenido.
visibilidad	pública

Método	
Nombre	setEstadoAnterior
Parámetros	double state[]
retorno	void
descripción	Guarda el estado “state” en la variable ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQStep
Parámetros	double state[],double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning utilizando “state” como estado siguiente, “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQEnd
Parámetros	double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning al finalizar cada episodio, utilizando “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	escribirTablaQ
Parámetros	void
retorno	void
descripción	Vuelca a fichero los valores de la tabla Q.
visibilidad	pública

Tabla 66: Métodos de la clase ACCSSWekaAgent.

Clase ACCSSWekaAgentLambda

Esta clase implementa el agente con el algoritmo de aprendizaje Q(λ), que hace uso de trazas de elegibilidad, utilizando una tabla Q por acción y, en consecuencia, un discretizador del espacio por cada acción. Como todo agente de la keepaway, esta clase implementa la interfaz SMDPAgent. Los atributos y métodos que la componen se presentan a continuación.

Tipo	Atributo	Descripción
WorldModel* (puntero)	WM	Puntero al modelo del mundo que proporciona información sobre el entorno en el que juega el agente, como por ejemplo el número de keepers y takers que están jugando.
FILE*	FICHERO	Puntero a fichero que es utilizado para volcar trazas de ejecución en forma de tuplas.
int	abierto	Indicador de si el fichero al que apunta FICHERO ha sido abierto
char [ ]	policy	Política utilizada por el controlador en caso de que el controlador implemente varias políticas de acción
int	OPCODE	Código entero que determina el nivel de detalle de trazado de la ejecución por salida std.
int	firstrew	Indica si es la primera vez que se ejecuta step o es la última del episodio antes de empezar el siguiente episodio.
FILE*	FICHERO_TABLA	Puntero al fichero que contiene las dimensiones y los valores de inicialización de la tabla Q.
double	CONST_A	Valor que toma la constante α en la función de actualización de la tabla Q.

double	CONST_B	Valor que toma la constante γ en la función de actualización de la tabla Q.
double	VAR_EPSILON	Valor que toma la variable ϵ en la política $\epsilon - greedy$.
double	VAR_INC_EPSILON	Valor en la que se incrementa la variable ϵ en la política $\epsilon - greedy$. El incremento se realiza al comienzo de cada episodio.
Char** (puntero doble)	NOMBREQTSAVE	Array con el Nombre de los ficheros en el que se vuelcan los valores de las tablas Q de forma regular.
Int* (puntero)	Q_TABLE_LENGTH	Array con el número de celdas que tienen cada una de las tablas Q (una por acción).
double** (puntero doble)	Q_TABLE	Puntero que apunta a las tablas Q.
int	QLEARNING	Variable que indica si se aplica el algoritmo Q-Learning sobre la tabla Q durante el juego. Q-Learning está activo solo cuanto esta variable toma el valor de 1.
int	ESCRIBIRQT	Indica cada cuantos episodios se debe volcar la tabla Q a fichero.
double	ESTADO_ANTERIOR	Puntero que apunta a la zona de memoria que guarda el estado anterior por el que transitó el agente.
int	LENGTH_ESTADO_ANTERIOR	Número de atributos que definen el estado anterior (y en consecuencia, cualquier estado del juego).
int	ACCION_ANTERIOR	Acción que el agente ejecutó en el estado anterior.
int	EPISODIOS	Número de episodios transcurridos desde el último volcado a fichero de la función Q.
int	START	Indica si el agente ha ejecutado la función startEpisode en el episodio actual.
double** (puntero doble)	Q_TABLE_VISITAS	Tablas de trazas de elegibilidad
double	delta	Valor delta de la función de actualización de trazas de elegibilidad.
double	elegibilidad	Valor de elegibilidad para el paso que se está tratando. Se guarda en esta variable pero se recupera originariamente de la tabla Q_TABLE_VISITAS
double	temp	Variable de uso temporal para elegibilidad.
double	f_q_ant	Valor anterior de la función Q para el paso que se está tratando.

double	nueva_elegibilidad	Variable donde se guarda temporalmente la actualización de la elegibilidad para el paso actual. Después se volcará a la tabla Q_TABLE_VISITAS.
double	lambda	Valor lambda de la función de actualización de trazas de elegibilidad.

Tabla 67: Atributos de la clase ACCSSWekaAgentLambda.

Método	
Nombre	choice
Parámetros	double state[],double reward
retorno	int
descripción	Devuelve la acción que mejor previsión de refuerzo ha obtenido para el estado “state”.
visibilidad	privada

Método	
Nombre	startEpisode
Parámetros	double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón por primera vez en el episodio actual. Recibe el estado actual y devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	step
Parámetros	double reward, double state[]
retorno	int
descripción	Se ejecuta cuando el agente obtiene el balón en el episodio actual excepto la primera vez. Recibe el estado actual y el tiempo que ha pasado desde que se deshizo del balón y lo ha vuelto a recuperar. Devuelve la acción elegida para ser ejecutada.
visibilidad	pública

Método	
Nombre	endEpisode
Parámetros	double reward
retorno	void
descripción	Se ejecuta cuando termina un episodio. Recibe el tiempo que ha pasado desde que se deshizo del balón y el momento en el que el episodio finaliza.
visibilidad	pública

Método	
Nombre	choiceMAXQ
Parámetros	double state[],double reward
retorno	double
descripción	Dado el estado “state” devuelve el mayor refuerzo que se puede conseguir desde ese estado. Más en detalle, calcula el refuerzo esperado para todas las acciones y devuelve el mayor valor obtenido.
visibilidad	pública

Método	
Nombre	setEstadoAnterior
Parámetros	double state[]
retorno	void
descripción	Guarda el estado “state” en la variable ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQStep
Parámetros	double state[],double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning utilizando “state” como estado siguiente, “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	actualizarTablaQEnd
Parámetros	double reward
retorno	void
descripción	Aplica la función de actualización Q-Learning al finalizar cada episodio, utilizando “reward” como refuerzo recibido después de aplicar ACCION_ANTERIOR en ESTADO_ANTERIOR.
visibilidad	pública

Método	
Nombre	escribirTablaQ
Parámetros	void
retorno	void
descripción	Vuelca a fichero los valores de la tabla Q.
visibilidad	pública

Tabla 68: Métodos de la clase ACCSSWekaAgentLambda.

La clase e interfaz TreeACCSS y las clases TreeACCSS_K

La interfaz TreeACCSS define la forma en que se comunican los agentes “ACCSSWekaAgent” y “ACCSSWekaAgentLamda” con los discretizadores del espacio de estados. Esta interfaz hace transparente el uso de los varios discretizadores (uno por acción) en la misma ejecución. La interfaz define el siguiente método que define como deben comportarse todos los discretizadores / aproximadores.

Método	
Nombre	treeACCSS
Parámetros	double state[],double reward,int action
retorno	int
descripción	Devuelve el índice de la tabla Q correspondiente al par estado-acción definido por “state” y “action”.
visibilidad	pública

Tabla 69: Método de la clase TreeACCSS.

Las distintas clases TreeACCSS_K definen cada una interfaz conocida por la Implementación de “TreeACCSS” e implementan el discretizador para la acción K. El método que define cada clase TreeACCSS_K se define a continuación.

Método	
Nombre	treeACCSS_K
Parámetros	double state[],double reward,int action
retorno	int
descripción	Devuelve el índice de la tabla Q de la acción K correspondiente al estado definido por el estado estado “state”. Tanto reward como action son innecesarios para obtener el resultado se les da un valor por defecto que será despreciable, por ejemplo, reward= -1 ,action= -1.
visibilidad	pública

Tabla 70: Método que implementa cada clase TreeACCSS_K, para K=0, 1 y 2.

Bibliografía

- [Albus, 1971] Albus, J.: *A theory of cerebellar function*. Mathematical Biosciences 1971; 10, 25-61

- [Albus, 1981] Albus J.: *Brain, Behaviour, and Robotics*. Byte Books, Peterborough, NH (1981)

- [Bellman, 1957] Bellman, R.: *Dynamic Programming*. Princeton University Press, Princeton, NJ. (1957)

- [Boyan y Moore, 1995] Boyan, J. A., Moore, A.W.: *Generalization in reinforcement learning: Safely approximating the value function*. Advances in Neural Information Processing Systems 1995; 7

- [Breiman et al, 1984] Breiman, L., Friedman, J.H., Olshen, R.A., Stone, C.J.: *Classification and Regression Trees*. Belmont, CA: Wadsworth, 1984.

- [Broomhead y Lowe, 1988] Broomhead, D.S., Lowe, D.: *Multivariable functional interpolation and adaptive networks*. Compex Systems (1988)

- [Duda y Hart, 1973] Duda, R.O., Hart, P.E.: *Pattern Classification and Scene Analysis*. John Wiley And Sons (1973)

- [Fernández y Borrajo, 2000] Fernández, F.; Borrajo, D.; (2000) *VQQL: Un Modelo de Aprendizaje por Refuerzo para Dominios Continuos e Indeterminísticos*. Proyecto Fin de Carrera. Universidad Carlos III de Madrid.

- [Fernandez, 2002] Fernandez, F.: *Aprendiezaje por Refuerzo en Espacios de Estados Continuos*. PhD thesis, Universidad Carlos III de Madrid, Leganés, Madrid, Spain (2002)

- [Fernández y Veloso, 2005] Fernández, F.; Veloso, M. (2005) *Exploration and Policy Reuse*. School of Computer Science. Carnegie Mellon University. Pittsburgh.

- [Fernández, 2006] Fernández, F. *Policy Reuse for Transfer Learning Across Tasks with Different State and Action Spaces.*

- [Fernandez y Borrajo, 2008] Fernandez, F., Borrajo, D.: *Two steps reinforcement learning*, Internation Journal of Intelligent Systems, Vol. 23, 213-245. 2008.

- [Frank y Witten, 1998] Frank, E., Witten, I.H.: *Generating Accurate Rule Sets Without Global Optimization*. Fifteenth International Conference on Machine learning, 144-151, 1998.

- [Frank y Witten, 2005] Frank, E., Witten, I.H.: *Data Mining: Practical Machine Learning Tools and Techniques* (Second Edition). Morgan Kaufmann. 2005.

- [Fritz et al, 1990] Fritz, W.; García Martínez, R. y Marsiglio, A.(1990). *Sistemas Inteligentes Artificiales*. Buenos Aires.

- [García et al, 2007] García, J.; Fernández, F.; Veloso, M.*: Reinforcement learning in the RoboCup-Soccer keepaway*. Conferencia de la Asociación para la Inteligencia Artificial (CAEPIA), Salamanca, 2007.

- [Gersho y Gray, 1992] Gersho, A., Gray, R.M.: *Vector Quantization and Signal Compression*. Kluwer Academic Publishers (1992)

- [Hinton, 1984] Hinton, G.E.: *Distributed representations*. Technical report, Department of Computer Science, Cargnegie Mellon University, Pittsburgh, PA (1984)

- [Kaelbling et al, 1996] Kaelbling, L.P., Littman, M.L., Moore, A.W.: *Reinforcement learning: A survey*. Journal of Artificial Intelligence Research 4 (1996) 237-285

- [Kohonen, 1989] Kohonen, T.: *Self-Organization and Associative Memory*. Sprinter, Berlin, Heidelberg (1984) 3rd ed. 1989.

- [Lloyd, 1957] Lloyd, S. P. (1957) *Least squares quantization in pcm*. Technical Note, Bell Laboratories, 1957. Publicado en 1982 en IEEE Transactions on Informacion Theory.

- [López-bueno et al, 2009] López-bueno, I.; García, J.; Fernández, F.: *Two Steps Reinforcement Learning in Continuous Reinforcement Learning Tasks*, 10th International Work-Conference on Artificial Neural Networks (IWANN 2009), Lecture Notes in Computer Science, Vol. 5517, 577-584. 2009.

- [Moriello, 2005] Moriello, S. (2005). *Inteligencia Natural y Sintética.* Buenos Aires, Editorial Nueva Librería.

- [Puterman, 1994] Puterman, M.L.: *Markov Decision Processes – Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY. (1994)

- [Quinlan, 1979] Quinlan, J.R.: *Discovering Rules by Induction from Large Collection of Examples*. Expert Systems in the Micro electronic Age. Ed. D. Michie. Edinburgh, UK. University Press, pp. 168-201, 1979.

- [Quinlan, 1986] Quinlan, J.R.: *Induction of Decision Trees*. Machine Learning 1 (1), pp. 81-106, 1986.

- [Quinlan, 1992] Quinlan, J.R.: *Learning with Continuous Classes*. 5th Australian Joint Conference on Artificial Intelligence, Singapore, 343-348, 1992.
- [Quinlan, 1993] Quinlan, J.R.: *Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

- [Sierra et al, 2006] Varios autores. Coordinador: Sierra, B.: Aprendizaje Automático: Conceptos básicos y avanzados. Pearson/Prentice Hall, 2006.

- [Soni y Singh, 2006] Soni, V.; Singh, S. (2006) *Using Homomorphisms to Transfer Options across Continuous Reinforcement Learning Domains*. Computer Science and Engineering. University of Michigan.

- [Stone et al, 2005] Stone, P.; Sutton, R.; Kulhmann, G. (2005) *Reinforcement Learning for RoboCup-Soccer Keepaway.*

- [Sutton y Barto, 1998] Sutton, R. y Barto A. (1998) *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

- [Taylor y Stone, 2005] Taylor, M.; Stone, P. (2005) *Behavior Transfer for Value-Function-Based Reinforcement Learning.* Department of Computer Sciences. The University of Texas at Austin.

- [Wang y Witten, 1997] Wang, Y., Witten, I. H.: *Induction of model trees for predicting continuous classes*. Poster papers of the 9th European Conference on Machine Learning, 1997.

- [Watkins, 1989] Watkins, C.J.C.H.: *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK (1989)